Technische Universität Dortmund
Fakultät Statistik

Automatische Modellselektion in der Überlebenszeitanalyse

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften

von

Dipl.-Stat.
MICHEL LANG

Vorgelegt: Dortmund, 4. März 2015
Tag der mündlichen Prüfung: 26. März 2015
Erstgutachter: Prof. Dr. Jörg Rahnenführer
Zweitgutachter: Dr. Bernd Bischl
Kommissionsvorsitz: Prof. Dr. Claus Weihs

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Dissertation selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Dissertation ist bisher keiner anderen Fakultät vorgelegt worden. Ich erkläre, dass ich bisher kein Promotionsverfahren erfolglos beendet habe und dass keine Aberkennung eines bereits erworbenen Doktorgrades vorliegt.

Michel Lang

Inhaltsverzeichnis

1 Einleitung

In den letzten ca. 15 Jahren hat sich ein großer Zweig der Krebsforschung auf die Analyse genetischer Daten konzentriert. Diese Daten entspringen Hochdurchsatz-Technologien, welche die gleichzeitige und kostengünstige Messung Tausender Gene erlauben. Die dabei erfassten Merkmale stellten in ihrem Volumen sowohl die Statistik als auch die Informatik vor neue Herausforderungen. So wurden zahlreiche neue Verfahren konzipiert, um die riesige Anzahl an Kovariablen verarbeiten zu können. Außerdem wurden viele bestehende Modelle und deren Implementierungen neu überdacht, da Laufzeitbetrachtungen und daraus resultierende Optimierungen der Ausführungszeit sich zuvor primär auf die Anzahl der Beobachtungen, nicht aber auf die Anzahl der Variablen konzentriert haben. Zudem rückte die Notwendigkeit der Parallelisierung gleichermaßen stärker in das Bewusstsein von Software-Entwicklern und Anwendern.

Einen der wohl größten Effekte in der Statistik hatten Hochdurchsatz-Technologien auf die Überlebenszeitanalyse. Dies liegt an dem klassischen Aufbau klinischer Studien und insbesondere von Krebsstudien, welche genetische Dispositionen berücksichtigen. Denn ein Behandlungserfolg ist hier nicht ohne Weiteres quantifizierbar: Von einer Heilung kann erst nach vielen Jahren ausgegangen werden, außerdem scheiden zahlreiche Patienten aus den mehrjährigen Studien aus nicht krankheitsrelevanten Gründen aus. Eine binäre Einteilung in wirksam gegen unwirksam bzw. geheilt gegen nicht geheilt ist so nicht möglich. Stattdessen kann lediglich beobachtet werden, wie lange ein Patient sich in der Studie befindet und ob für das Ausscheiden ein relevantes Ereignis ursächlich ist oder nicht. Die Klassifikations- oder Regressionsanalyse sowie statistische Hypothesentests stellen bei Vorliegen von solchen zensierten zeitlichen Daten gegenüber den Methoden der Überlebenszeitanalyse keine sinnvollen Alternativen dar.

Das Ziel der Überlebenszeitanalyse ist dabei oft zweigeteilt. Die Berücksichtigung genetischer Dispositionen ist zur Prognose des Krankheitsverlaufes wichtig und kann Kliniker bei der Therapiewahl unterstützen. Eine Einteilung in Risikogruppen kann dabei aber nur auf angemessenen Modellen basieren, weshalb das vornehmliche Ziel hier die Wahl und

Konfiguration von Modellen mit ordentlicher prädiktiver Güte ist. Neben der Prädiktionsleistung ist aus biologischer und pharmazeutischer Sicht aber oft auch die Identifikation von jenen Genen wichtig, welche den Krankheitsverlauf entweder mildern oder beschleunigen. Ein statistisch auffälliges Gen ist so Ausgangspunkt zur weiteren Untersuchung seiner biologischen Funktionalität und gegebenenfalls anschließender zielgerichteter Entwicklung neuer Pharmaka. Die genetische Analyse ermöglicht so eine personalisierte Behandlung. In der Praxis sind beide Ziele jedoch nicht voneinander trennbar, da eine Extraktion interessanter Gene nur aus performanten Modellen sinnvoll ist. Implizit handelt es sich also um ein multikriterielles Problem, welches jedoch nicht als solches formuliert wird.

Während die Hochdurchsatz-Daten aufgrund verbesserter Auflösung und neuer Messtechnologien immer umfangreicher werden und viele Datensätze auf Plattformen wie der *Gene Expression Omnibus* (GEO) Datenbank (Edgar, Domrachev und Lash, 2002) veröffentlicht wurden, haben sich gleichermaßen auch die statistischen Modelle und Ansätze zur Analyse weiterentwickelt. Jedoch fehlt zur statistischen Nutzbarkeit in der Praxis immer noch ein entscheidender Bestandteil: Denn ohne das Wissen, (a) welche Modelle (b) mit welchen Hyperparametern und (c) welcher Vorverarbeitung für welchen der beiden Zwecke zuverlässig auf der ganzen Domäne einer Krebsart funktionieren, ist eine sinnvolle statistische Analyse quasi unmöglich. Es fehlt also an umfangreichen Benchmarkstudien. Diese erfordern allerdings auch das Zusammenspiel vieler Kompetenzen. Neben der Überlebenszeitanalyse spielt die Bioinformatik zur Vorverarbeitung der Rohdaten eine wichtige Rolle. Zudem haben die Modelle sehr hohe Ressourcenanforderungen an CPU-Zeit und Arbeitsspeicher, was eine Konfiguration der Modelle erheblich erschwert und ineffiziente Heuristiken wie etwa eine Gittersuche ausschließt. Stattdessen empfehlen sich moderne Verfahren zur Hyperparameteroptimierung, welche mit einer begrenzten Anzahl an Evaluationen zuverlässig gute Einstellungen für Vorverarbeitung und Modelle bestimmen. Nicht zuletzt ist auch die Softwareentwicklung und insbesondere die Parallelisierung elementar, da solche Benchmarkstudien oft nicht trivial programmierbar sind und sich sequentielle Rechenzeiten schnell zu mehreren Jahren summieren können.

In dieser Arbeit wird ein Ansatz zur vollautomatischen Modellselektion auf mehreren Lungenkrebsdatensätzen evaluiert. Dazu wird ein Meta-Lernverfahren definiert, welches je nach Parametrisierung eine Variablenselektion durchführt und anschließend ein Überlebenszeitmodell auf den verbleibenden Daten anpasst. Der Raum aller Hyperparameter, sowohl die der Filter als auch Modelle, wird dabei zusammen betrachtet. Dabei stellen die Wahl der Variablenselektion sowie die Wahl des Modells ebenfalls Hyperparameter dar. Die Aufgabe des Optimierers ist es nun, jenen Punkt in diesem Raum zu identi-

fizieren, welcher zu der Parametrisierung mit minimalem Vorhersagefehler führt. Der Aspekt der gewünschten Modell-Spärlichkeit wird auf zwei Arten bedacht: Einerseits schließt das Modell-Portfolio viele Modelle mit eingebetteter Variablenselektion ein, anderseits werden zudem Variablenselektionsfilter vorgeschaltet. Nicht-spärliche Lösungen sind jedoch ebenfalls zulässig, eine Bestrafung basierend auf der Anzahl verwendeter Kovariablen wird nicht betrachtet. Hauptaugenmerk liegt demnach auf der Prädiktionsleistung. Zur Modellselektion wurde Unterstützung für die Überlebenszeitanalyse in das `mlr`-Framework (Bischl u. a., 2014a) für Maschinelles Lernen eingearbeitet. Dadurch ist ein generischer Ansatz für Benchmarkstudien in der Überlebenszeitanalyse entstanden. Außerdem ist zur Parallelisierung umfassend Software entwickelt worden, so dass die Modellselektion ohne Anpassungen an das Ausführungssystem auf einer Vielzahl von Systemen dort verfügbare Ressourcen effizient nutzt.

Im nächsten Kapitel 2 wird zunächst der aktuelle Stand der Forschung reflektiert. Die Theorie der Überlebenszeitanalyse wird im anschließenden Kapitel 3 erläutert. Neben den wichtigsten grundlegenden Begriffen und Modellen umfasst dies auch die Variablenselektion durch Filter sowie Gütemaße für die Überlebenszeitanalyse. Die Optimierung für teure Black-Box Probleme wird in Kapitel 4 thematisiert und dabei die Brücke zur Hyperparameteroptimierung geschlagen. Entwickelte Lösungen zum parallelen wissenschaftlichen Rechnen werden in Kapitel 5 vorgestellt. Die Anwendung der Optimierung findet sich in Kapitel 6, wobei die durch Optimierung gefundenen Modelle mit einer Vielzahl konkurrierender Ansätze ausführlich verglichen werden. Kapitel 7 schließt mit einer Zusammenfassung der Arbeit ab.

2 Aktueller Stand der Forschung

Das Repertoire der Überlebenszeitanalyse wurde mit steigender Verfügbarkeit von Hochdurchsatz-Daten schnell um klassische Ansätze aus der Regressionsanalyse erweitert und damit für die hohe Dimensionalität adaptiert. Dies umfasst etwa die Konzepte des Boostings (Binder und Schumacher, 2008a; Hothorn u. a., 2010), der Penalisierung (Verweij und Van Houwelingen, 1994; Bühlmann und Yu, 2003; Zou und Hastie, 2005; Goeman, 2010; Benner u. a., 2010) oder der rekursiven Partitionierung durch Bäume (LeBlanc und Crowley, 1993; Hothorn u. a., 2004; Hothorn u. a., 2006; Ishwaran u. a., 2008). Im Hinblick auf das duale Ziel, Variablenselektion bei gleichzeitig hoher Prädiktionsgüte, wurden favorisiert Methoden aus anderen Bereichen übertragen, welche eine eingebettete Variablenauswahl beinhalten (also spärliche Lösungen produzieren) oder zumindest ein einfaches Ranking von Genen ermöglichen. Jedoch finden auch die Vorauswahl von Genen durch Filter oder dimensionsreduzierenden Techniken wie die Hauptkomponentenanalyse, gefolgt von einer Modellanpassung, oft Beachtung. Ein erster umfangreicher Vergleich dieses Vorgehens findet sich in Bøvelstad u. a. (2007), wo univariate Suche, Vorwärtssuche, überwachte und unüberwachte Hauptkomponentenanalyse, partielle Kleinste-Quadrate-Regression, und L_1- und L_2-Regression untersucht werden. In Witten und Tibshirani (2009) wird das Portfolio etwa durch *Sliced inversed regression* oder eine Hauptkomponentenregression erweitert. In beiden Arbeiten stellen sich penalisierte Cox Proportional Hazard-Modelle (Cox, 1972) den anderen Methoden meist hinsichtlich ihrer Güte als überlegen heraus. Dabei ist die Lasso-Regression (Tibshirani, 1996; Tibshirani, 1997) der Ridge-Regression (Hoerl und Kennard, 1970) eher unterlegen, bietet andererseits jedoch durch ihre implizite Variablenauswahl eine spärliche Lösung und somit direkt die gewünschte Dimensionsreduktion. Diese Ergebnisse wurden im Wesentlichen in Kammers u. a. (2011) bestätigt, wobei hier eine Anreicherung der Daten mit zusätzlichen Informationen aus Annotations-Datenbanken zu einer besseren biologischen Interpretierbarkeit führt. De Bin, Sauerbrei und Boulesteix (2014) bieten eine aktuellere Übersicht mit ähnlich umfangreichem Methoden-Portfolio. Die Wichtigkeit der Variablenselektion wird unter anderem auch durch die Arbeit in Lee u. a. (2014) unterstrichen, wo spezielle Lasso-

Regressionen zum Einsatz kommen und die Lösungen neben der prädiktiven Leistung auch anhand der Robustheit ihrer Variablenselektion über mehrere Kreuzvalidierungen hinweg beurteilt werden.

Obwohl also einige Vergleichsstudien existieren, mangelt es immer noch an einem systematischen, umfangreichen und damit vertrauenswürdigen Vergleich. Alle genannten Studien, und viele weitere Studien dieser Art, wurden auf maximal vier Datensätzen validiert, meist sind es zwei oder vereinzelt auch nur ein einziger Datensatz. Dies liegt sowohl an der Datenverfügbarkeit als auch der fehlenden Spezifikation eines Formates. Da sich kein maschinenlesbares Format für Überlebenszeitdaten etabliert hat, müssen hochdimensionale Datensätze händisch heruntergeladen, konvertiert und bereinigt werden. Wichtige klinische Zusatzinformationen sind auch in Datenbanken wie GEO (Edgar, Domrachev und Lash, 2002) nicht immer direkt verfügbar, sondern müssen separat angefordert werden. Teilweise werden klinische Daten sogar bewusst zurückgehalten. Aber auch die Heterogenität der Datensätze ist mitverantwortlich für das Fehlen von größeren Vergleichsstudien. Beispielsweise definiert Hudis u. a. (2007) verschiedene Endpunkte für Brustkrebs und illustriert dabei anschaulich die mangelnde Standardisierung in klinischen Studien. Wenn jedoch sogar die Zielvariable verschieden interpretiert und aufgefasst wird, ist eine gepoolte Analyse mehrerer Datensätze sehr schwierig. Eine Diskussion über die Möglichkeiten und Probleme, mehrere einzelne Datensätze aus verschiedenen Kliniken oder Studien zu fusionieren, wurde bereits in Järvinen u. a. (2004) angestossen.

Darüberhinaus spielen auch die Laufzeiten eine große Rolle. Da eine einzelne Modellanpassung mehrere Stunden dauern kann, werden Methoden auf entsprechend wenigen Datensätzen evaluiert. Teilweise wird dabei sogar nur eine einzige Aufteilung in Trainings- und Testmenge validiert, etwa in Witten und Tibshirani (2009) oder De Bin, Sauerbrei und Boulesteix (2014). Ein wirklicher Erkenntnisgewinn ist unter diesen Voraussetzungen selbstverständlich ausgeschlossen. Denn ob es sich bei den Beobachtungen in diesen Studien um Artefakte handelt, oder ob eine Überanpassung an die Testmenge vorliegt, ist vollkommen unklar. Jedoch werden trotzdem häufig Aussagen aus den Beobachtungen abgeleitet.

In einem Punkt sind sich jedoch alle Übersichtsarbeiten relativ einig: Ein Modell zu finden, welches besser ist als ein niedrigdimensionales Modell basierend auf den klinischen Daten, ist eine Herausforderung.

Erschwerend kommt hinzu, dass die Gütemaße oft umstritten, schwer miteinander vergleichbar und sich oft „uneinig“ sind: Wird einem Verfahren von einem Maß eine hohe

Vorhersagegüte attestiert, so ist oft zu beobachten, dass ein anderes Maß dasselbe Verfahren als nicht unterscheidbar von zufälligen Prognosen einstuft. Auch gegensätzliche Einschätzungen sind nicht ungewöhnlich: Falls ein Gütemaß ein Modell A dem Modell B überlegen bewertet, kann ein anderes Maß Modell A als unterlegen einstufen. Durch die Einführung von Parametern für einige Gütemaße hat sich die Situation noch weiter verschlimmert: Sogenannte *Cutoff*-Parameter können beispielsweise eingestellt werden, um Prognosen nach einem gewissen Zeitintervall vollständig zu ignorieren. Dadurch wird ein Datensatz unnötig „leichter“ oder „schwieriger“ und die Vergleichbarkeit mit anderen Studien geht verloren. Außerdem gelten Schranken für Zufälligkeit nicht mehr: Die Grenzwerte $\frac{1}{3}$ und $\frac{1}{4}$ des Brier-Scores (Brier (1950), siehe auch Abschnitt 3.2.2), verschieben sich unbestimmbar, so dass die Prädiktionsgüte leicht überschätzt werden kann. Viele andere Gütemaße, welche primär Anfang bis Mitte der 2000er genutzt wurden, fußen auf der Likelihood-Funktion des Cox Proportional Hazard-Modells. Der bekannte Score-Test ist ein prominentes Beispiel, andere sind in Verweij und Van Houwelingen (1993) und Tibshirani und Efron (2002) zu finden. Eine mehrmalige Aufteilung in Trainings- und Testmenge soll so vermieden werden. Jedoch vernachlässigen diese Maße die Tatsache, dass falls die Modellvoraussetzungen nicht halten, die Interpretation der Gütemaße als Interpretation einer misspezifizierten Likelihood irreführend sein kann. Weiterhin kann eine gewisse Überanpassung hier nicht ausgeschlossen werden, da eine Überanpassung letztendlich nur durch eine Validierung auf unabhängigen Daten sichergestellt werden kann.

Die Optimierung der Parameter der Vorverarbeitung, Variablenauswahl und Modellierung wurden bei allen Arbeiten bisher vernachlässigt. Einige Implementierungen bieten die Optimierung weniger ausgewählter Parameter an, was auch in einigen der genannten Übersichtsarbeiten genutzt wird. Jedoch deckt dies in der Regel nicht alle Parameter einer Implementierung und sicher nicht alle Parameter eines Experiments ab. Insbesondere Parameter der Variablenauswahl wurden stets a-priori festgelegt. Dies ist erneut mit dem großen rechnerischem Aufwand verbunden, den eine halbwegs vollständige Parameteroptimierung mit sich brächte. Jedoch hat die Black-Box-Optimierung in den letzten Jahren große Fortschritte gemacht und bietet hier Auswege. Für diskrete Parameterräume bietet das Racing (Birattari u. a., 2002) eine effiziente Herangehensweisen, für numerische Parameterräume stellt die *Efficient Global Optimization* (EGO) (Jones, Schonlau und Welch, 1998) eine Lösung bereit. Für gemischt-skalierte Parameterräume wurde das Racing zum *Iterated F-Racing* (López-Ibánez u. a., 2011) erweitert und in Lang u. a. (2015) bereits auf Überlebenszeitmodelle angewendet. Beim EGO können

einige module Komponenten ausgetauscht werden, was ebenfalls eine Erweiterung auf gemischt-skalierte Räume ermöglicht. Diese Art der Verallgemeinerungen fallen unter den Begriff der *Model Based Optimization* (MBO), was explizit aber auch EGO als Spezialfall mit einschließt. Eine MBO-Implementierung für Klassifikationsprobleme ist durch Auto-WEKA (Thornton u. a., 2013) gegeben. Durch die Verwendung dieser modernen Optimierer kann im Vergleich mit einer Gittersuche eine bessere Modellkonfiguration in weniger Schritten gefunden werden. Dennoch bleibt der rechnerische Aufwand für Überlebenszeitdaten so hoch, dass eine in den Optimeriungsprozess nativ unterstützte Parallelisierung unausweichlich ist. Die Parallelisierung richtet sich dabei primär nach der Architektur des Zielsystems. Für die Forschung stellen derzeit *high-performance computing cluster* (HPCs) die vermeintlich wichtigste Plattform dar. Eine erstmalige umfangreiche Unterstützung mehrerer Scheduler-Dialekte in `R` (R Core Team, 2014) ist durch das Paket `BatchJobs` (Bischl u. a., 2012a) verwirklicht. Dabei ist die Parallelisierung nicht nur ein technisches, sondern auch ein methodisches Problem. Die verschiedenen Strategien zur Parallelisierung der MBO werden in Bischl u. a. (2014c) ausführlich untersucht. Eine Anwendung der Hyperparameteroptimierung ist etwa in Bischl u. a. (2012a) zu finden, in der simultan numerische Parameter der Vorverarbeitung und Klassifikation optimiert werden, parallelisiert durch das Paket `BatchExperiments` (Bischl u. a., 2012a).

3 Methoden der Überlebenszeitanalyse

In der Überlebenszeitanalyse wird die Zeit bzw. Dauer bis zum Eintreten eines zuvor festgelegten Ereignisses betrachtet. In Abgrenzung zur Regression können hier jedoch Zensierungen in der Zielvariablen auftreten – anstatt also den genauen Wert zu kennen, ist lediglich bekannt, dass das Ereignis entweder vor einem bestimmten Zeitpunkt eingetreten sein muss (Linkszensierung) oder bis zu einem Zeitpunkt noch nicht eingetreten ist (Rechtszensierung). Die Kombination aus beiden Zensierungsarten wird als Intervallzensierung bezeichnet. In dieser Arbeit werden ausschließlich rechtszensierte Überlebenszeiten betrachtet, welche in der Praxis aufgrund des klassischen Design klinischer Studien auch am häufigsten erhoben werden. Patienten werden in klinischen Studien in einer Rekrutierungsphase akquiriert, ihr Zeitpunkt des Studienbeitritts ist damit stets bekannt. Unterschiede in den Eintrittszeitpunkten von wenigen Wochen können in der Regel ignoriert werden, da hier kein zeitlicher Effekt, welcher beispielsweise auf fortgeschrittenere Behandlungsmethoden zurückzuführen wäre, zu erwarten ist. Letztlich wird daher der Eintrittszeitpunkt für alle Patienten als identisch betrachtet und es interessiert lediglich die Dauer zum Eintreten eines zuvor definierten Ereignisses, welches unter Umständen nicht immer zu beobachten ist. Ein klassisches Beispiel stellt hier das Überleben von Krebspatienten dar, wobei der Tod das Ereignis darstellt. Verlässt ein Patient die Studie, beispielsweise weil er in eine andere Stadt zieht, so liegt eine Zensierung vor. Die Information, dass der Patienten bis zum Ausscheiden aus der Studie noch am leben war, stellt jedoch eine hilfreiche Information dar.

In diesem Kapitel werden zunächst in Abschnitt 3.1 die zentralen Begriffe und Funktionen der Überlebenszeitanalyse erläutert. Anschließend werden in Abschnitt 3.2 Maße zur Beurteilung der Prädiktionsgüte von Überlebenszeitmodellen vorgestellt. In Abschnitt 3.3 werden Variablenselektionsfilter betrachtet. Abschnitt 3.4 thematisiert Überlebenszeitmodelle basierend auf dem Cox Proportional Hazard-Modell, im abschließenden Abschnitt 3.5 wird mit Überlebenszeitbäumen und Überlebenszeitwäldern eine alternative Modellierungsstrategie vorgestellt.

3.1 Zentrale Begriffe

Formal werden zwei nicht-negative Zufallsvariablen betrachtet: Die Zeit Z bis zum Eintreten des Ereignisses sowie die Zeit C bis zum Eintreten einer Zensierung. Beobachtet werden kann davon nur $T = \min(Z, C)$, die Zeit bis entweder ein Ereignis oder eine Zensierung auftritt. In dieser Arbeit wird davon ausgegangen, dass C keinerlei Informationen über Z enthält, was unter anderem stochastische Unabhängigkeit impliziert. Diese Voraussetzung wird als *nicht informative Zensierungen* bezeichnet. Die n-dimensionalen Kovariablenvektoren $\boldsymbol{x}_j$ $(j = 1, \ldots, p)$ werden in der Matrix $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_p)$ zusammengefasst. Dabei lassen sich die Kovariablen für diese Arbeit sinngemäß in zwei Gruppen aufteilen: Einerseits liegen gemischt-skalierte klinische Kovariablen wie etwa Alter, Geschlecht, Tumorgrad oder Tumorgröße vor. Andererseits gibt es mehrere Zehntausend genetische Kovariablen, welche die Aktivität von Genen metrisch repräsentieren.

Der zusammengesetzte Raum aus klinischen und genetischen Kovariablen wird mit $\mathcal{X}$ bezeichnet. Ein Element aus einer Lernstichprobe $\mathcal{L}$ setzt sich aus dem Tripel $(t_i, \delta_i, \boldsymbol{x}_i)$ zusammen, wobei $\delta = \mathbb{1}(Z \leqslant C)$ den Zensierungsindikator bezeichnet. Falls erforderlich, bezeichnen $\mathcal{X}_c$ und $\mathcal{X}_g$ die Untermenge der klinischen respektive genetischen Kovariablen. Die Entscheidung zur Berücksichtigung klinischer Kovariablen ist nicht selbstverständlich, sondern richtet sich nach den gesteckten Zielen. Steht die Modellierung zur Vorhersage anstatt der Interpretation und Identifikation verwendeter Kovariablen im Vordergrund, sorgt die Aufnahme klinischer Kovariablen in der Regel für eine Steigerung der Prognosegüte. Werden die klinischen Variablen darüber hinaus speziell behandelt, etwa als Startmodell (*Offset*) in Boosting-Ansätzen oder durch Exklusion vom Strafterm in penalisierten Modellen, wird von „zusätzlichem Nutzen“ (*added value*) gesprochen. Verwendete zusätzliche Kovariablen sind in diesem Fall zumeist unkorrelliert von den beobachteten klinischen Faktoren und somit von besonderem (im Sinne von anders nicht identifizierbar) medizinischen Interesse. Liegt der Fokus dagegen primär auf der Identifikation krankheitsrelevanter Gene, können sich durch die Modellierung ausschließlich basierend auf genetischen Kovariablen auch solche Kovariablen herauskristallisieren, welche mit klinischen Faktoren zwar korreliert sind, aber dennoch als Bestandteil eines biologische Prozesses den Krankheitsverlauf nachhaltig beeinflussen können.

In dieser Arbeit steht der erste Fall, die Optimierung der Vorhersagegüte, im Vordergrund. Gleichzeitig ist dieser Schritt jedoch auch Voraussetzung für den zweiten Schritt, da ohne ein gewisses Maß an Modellgüte eine Analyse des Modells oder eine genaue Betrachtung selektierter Variablen wenig Sinn macht.

3.1.1 Survivalfunktion

Die zentrale Funktion der Überlebenszeitanalyse stellt die Überlebenszeitfunktion oder Survivalfunktion

$$S(t) = P(T > t) = 1 - F(t)$$

dar, also die Wahrscheinlichkeit, einen Zeitpunkt t zu überleben. Die Survivalfunktion $S(t)$ ist, wie durch den Zusammenhang mit der Verteilungsfunktion $F(t)$ von T ersichtlich, rechtsseitig stetig und monoton fallend mit den Grenzwerten $S(t) = 1$ für $t \to -\infty$ und $S(t) = 0$ für $t \to \infty$.

Zur Konstruktion eines Schätzers werden die paarweise verschiedenen und geordneten Ereigniszeiten $t_{(i)}$ $(i = 1, \ldots, T)$ mit $t_{(i)} < t_{(i+1)}$ benötigt. Außerdem ist die zum Zeitpunkt $t_{(i)}$ zugehörige Anzahl d_i der aufgetretenen Ereignisse erforderlich, wobei eine Zensierung hier kein Ereignis darstellt. Die Anzahl der Beobachtungen, welche zum Zeitpunkt $t_{(i)}$ noch unter Risiko stehen, also für die weder ein Ereignis noch eine Zensierung aufgetreten ist, wird mit n_i bezeichnet. Damit ist nach Kaplan und Meier (1958) durch

$$\hat{S}(t) = \begin{cases} 1, & t < t_{(1)} \\ \prod_{t_{(i)} \leqslant t} \left(1 - \frac{d_i}{n_i}\right), & t \geqslant t_{(1)} \end{cases}$$

ein konsistenter Schätzer, der Kaplan-Meier-Schätzer, definiert. Der zugehörige Standardfehler kann nach Klein und Moeschberger (2003) durch die Formel von Greenwood mittels

$$\text{se}\left(\hat{S}(t)\right) = \hat{S}(t) \cdot \sqrt{\sum_{t_{(i)} \leqslant t} \frac{d_i}{n_i(n_i - d_i)}}$$

angegeben werden. Das Grenzwertverhalten des Kaplan-Meier-Schätzers entspricht dem Grenzwertverhalten der theoretischen Survivalfunktion nur für den Fall, dass der letzte Beobachtungszeitpunkt zensierungsfrei ist. Andernfalls verbleibt $\hat{S}(t)$ für $t \in [t_{(n)}, \infty)$ auf einem konstanten Wert $c > 0$.

3.1.2 Hazardrate

Eine weitere wichtige Kenngröße stellt die Hazardrate dar (Klein und Moeschberger, 2003). Sie beschreibt das Risiko, genau zu einem Zeitpunkt z auszufallen, konditioniert

auf die Bedingung, bis zu dem Zeitpunkt z überlebt zu haben:

$$\lambda(z) = \lim_{h \downarrow 0} \frac{P\left(z \leqslant Z < z + h \mid Z > z\right)}{h}, \qquad \lambda(z) \geqslant 0 \ \forall z \in [0, \infty].$$

Für stetiges Z kann der Zusammenhang zwischen Hazardrate und Survivalfunktion durch

$$\begin{aligned} \lambda(z) &= \lim_{h \downarrow 0} \frac{P\left(z \leqslant Z < z + h\right)}{h} \frac{1}{P\left(Z > z\right)} \\ &= \frac{f(z)}{S(t)} = \frac{\frac{\partial}{\partial z} F(z)}{S(z)} = \frac{-\frac{\partial}{\partial z} S(z)}{S(z)} = -\frac{\partial}{\partial z} \ln\left(S(z)\right) \end{aligned}$$

ausgedrückt werden. Der populäre Logrank-Test zu dem Testproblem

$$H_0 \colon \lambda_1(t) = \lambda_2(t) \ \forall t \quad \text{vs.} \quad H_1 \colon \exists t : \lambda_1(t) \neq \lambda_2(t)$$

ist in Mantel (1963) formuliert und basiert auf der hypergeometrischen Verteilung. Seien N_{1j} und N_{2j} die Anzahl der Beobachtungen unter Risiko zum Ereigniszeitpunkt t_j und $N_i = N_{1i} + N_{2i}$ die Summe der Beobachtungen unter Risiko in beiden Gruppen. Seien analog O_{1j} und O_{2j} die Anzahl der beobachteten Ereignisse in beiden Gruppen und O_j deren Summe. Dann ist O_{1j} hypergeometrisch Verteilt mit Parametern (N_j, N_{1j}, O_j). Mit Erwartungswert $E_{1j} = \frac{O_j}{N_j} N_{1j}$ und Varianz $V_j = \frac{O_j (N_{1j}/N_j)(1 - N_{1j}/N_j)(N_j - O_j)}{N_j - 1}$ ergibt sich die unter der Nullhypothese asymptotisch standardnormalverteilte Teststatistik

$$Z = \frac{\sum_j \left(O_{ij} - E_{ij}\right)}{\sqrt{\sum_j V_j}}.$$

3.1.3 Kumulative Hazardrate

Die dritte zentrale Kenngröße ist die kumulative Hazardrate

$$\Lambda(z) = \int_0^z \lambda(u)\,\mathrm{d}u = \int_0^z \frac{f(u)}{S(u)}\,\mathrm{d}u = \int_0^z \frac{\frac{\delta}{\delta u} F(u)}{S(u)}\,\mathrm{d}u = -\int_0^z \frac{\frac{\delta}{\delta u} S(u)}{S(u)}\,\mathrm{d}u = -\ln S(z).$$

Ein Schätzer ist in den Arbeiten von Nelson (1972) und Aalen (1978) gegeben. Stellen erneut $t_{(i)}$, $i = 1, \ldots, D$ mit $t_{(1)} < t_{(2)} < \cdots < t_{(D)}$ die geordneten und eindeutigen Ereigniszeiten dar, so kann wie zuvor zu jeder Ereigniszeit $t_{(i)}$ die Anzahl der Ereignisse d_i sowie die zugehörige Patientenanzahl unter Risiko n_i unmittelbar vor dem

Zeitpunkt $t_{(i)}$ betrachtet werden. Der Nelson-Aalen-Schätzer ergibt sich damit nach Klein und Moeschberger (2003) als

$$\hat{\Lambda}(t) = \sum_{t_{(i)} \leqslant t} \hat{\lambda}(t_{(i)}) = \sum_{t_{(i)} \leqslant t} \frac{d_i}{n_i}.$$

Entsprechend dem Zusammenhang zwischen Survivalfunktion $S(t)$ und kumulativer Hazardrate $\Lambda(t)$ ist durch

$$\hat{S}(t) = \exp\left(-\hat{\Lambda}(t)\right) = \exp\left(-\sum_{t_{(i)} \leqslant t} \frac{d_i}{n_i}\right)$$

ein Schätzer für die Survivalfunktion durch den Nelson-Aalen-Schätzer gegeben.

3.2 Gütemaße

Die hier vorgestellten Gütemaße sind stets im Kontext eines Resamplingverfahrens wie beispielsweise einer Kreuzvalidierung zu sehen. Entsprechend wird eine Lernstichprobe $\mathcal{L}$ des Umfangs n aus der Gesamtheit gezogen. Auf diesen Trainingsdaten wird dann entweder eine zeitunabhängige Risikofunktion $r : \mathcal{X} \to \mathbb{R}$ oder alternativ eine Überlebenszeitfunktion $\hat{S}(t \mid \boldsymbol{x}_i)$ in Abhängigkeit von Zeit und Kovariablen gelernt. Diese werden anschließend auf unabhängigen Testdaten evaluiert. In dieser Arbeit wird ausschließlich die Kreuzvalidierung genutzt, einen Überblick über alternative Resampling-Verfahren bieten Molinaro, Simon und Pfeiffer (2005) und Bischl, Mersmann und Trautmann (2010). Dabei ist zu beachten, dass viele Überlebenszeitmodelle mit duplizierten Beobachtungen, wie sie etwa beim Bootstrap auftreten, nicht kompatibel sind.

3.2.1 Konkordanzindex

Der Konkordanzindex, auch C-Index genannt, bewertet die Übereinstimmungen der Rangfolge einer Risikofunktion r mit der Rangfolge der tatsächlich beobachteten Überlebenszeiten. Wenn also trotz Zensierungen für zwei Stichprobenelemente mit Überlebenszeiten t_i und t_j festgestellt werden kann, dass $t_i \lesseqgtr t_j$ gilt, so gilt das Paar (t_i, t_j) als konkordant genau dann, wenn $r(\boldsymbol{x}_i) \gtreqless r(\boldsymbol{x}_j)$. Die Vektoren $\boldsymbol{x}_i$ und $\boldsymbol{x}_j$ sind dabei die bereits in der

Einleitung definierten Kovariablenvektoren einer Testmenge. Formal lässt sich das nach Heagerty und Zheng (2005) durch

$$C(t, \boldsymbol{x}) = \frac{1}{|\epsilon|} \sum_{\{i:\ \delta_i=1\}} \sum_{\{j:\ t_j>t_i\}} \left(\mathbb{1}\left(r(\boldsymbol{x}_i) < r(\boldsymbol{x}_j)\right) + \frac{1}{2}\mathbb{1}\left(r(\boldsymbol{x}_i) = r(\boldsymbol{x}_j)\right) \right)$$

ausdrücken. Dabei stellt ϵ die Menge aller vergleichbaren Kombinationen von (i, j) dar, deren Mächtigkeit $|\epsilon|$ normiert somit auf das Intervall $[0, 1]$. Für $C \approx 1$ kann von einer sehr guten Prädiktion ausgegangen werden, Werte um 0.5 sprechen für zufällige Vorhersagen und $C \approx 0$ für gegensätzliche Vorhersagen.

3.2.2 Brier Score

Der Brier-Score (Brier, 1950) stellt das Analogon zu dem mittleren quadratischen Fehler der linearen Regressionsanalyse dar. Seien $\hat{S}(t)$ und $\hat{G}(t)$ die Kaplan-Meier-Schätzer der Überlebenszeit bzw. der Zensierungszeit. Dann lässt sich der quadratische Verlust zwischen vorhergesagter Ausfallwahrscheinlichkeit $\hat{S}(t \mid \boldsymbol{x}_i)$ und der wahren Ausfallverteilung durch

$$\mathrm{BS}(t) = \frac{1}{n} \sum_{i=1}^{n} \left(\mathbb{1}\left(t_i > t\right) - \hat{S}\left(t \mid \boldsymbol{x}_i\right) \right)^2 w_i(t)$$

ausdrücken. Die Gewichtsfunktion

$$w_i(t) = \frac{\mathbb{1}\left(t_i < t\right) \delta_i}{\hat{G}(t_i)} + \frac{\mathbb{1}\left(t_i > t\right)}{\hat{G}(t)}$$

korrigiert durch Gewichtung mit der inversen Zensierungswahrscheinlichkeit dabei durch Zensierungen verursachte Verzerrungen (Schumacher, Binder und Gerds, 2007). Durch Integration über die Zeit bis zu einem maximalen Zeitpunkt t^* ergibt sich der integrierte Brier-Score

$$\mathrm{IBS}(t^*) = \frac{1}{t^*} \int_0^{t^*} \mathrm{BS}(t)\, \mathrm{d}t$$

als Maßzahl zur Beurteilung der Vorhersagegüte. Oft wird für t^* ein Wert $t^* < \max\left(t_i\right)$ gewählt. Dies resultiert darin, dass späte Beobachtungen weniger Beachtung finden, hat aber eine Reihe von Nachteilen:

- Der IBS kann in der Praxis kleiner (optimistischer) werden, die Ergebnisse lassen sich so leicht überinterpretieren.

- Eine obere Schranke für Zufälligkeit kann nicht mehr eindeutig spezifiziert werden (ansonsten IBS = 0.25).
- Der Vergleich über mehrere Datensätze hinweg ist unmöglich, wenn t^* abhängig von den Daten gewählt wird.

Da das Modell zeitabhängige Prognosen der Ausfallwahrscheinlichkeit liefern muss, gestaltet sich die Evaluation mit dem Brier-Score für manche Modellklassen schwierig oder unintuitiv, etwa für Überlebenszeitbäume. Für einen Vergleich über mehrere Modellklassen hinweg wird daher in der Regel der einfachere Konkordanzindex verwendet, bei dem lediglich ein Rangfolge der Beobachtungen zu extrahieren ist.

3.3 Filter

Filter spielen in der hochdimensionalen Überlebenszeitanalyse eine wichtige Rolle, da ein einem Modell vorgeschalteter Filter mehrere positive Effekte bewirken kann: (a) Es lassen sich Methoden anwenden, welche „$n > p$“ voraussetzen, (b) die Güte des Modells kann gesteigert werden, und (c) die Laufzeit der anschließenden Modellanpassung wird drastisch reduziert. Natürlich kann das Filtern auch negative Effekte haben, schließlich werden dem Modell potentiell wichtige Informationen vorenthalten. Deshalb ist es hier vorteilhaft, den Grad der Filterung, also wieviele Variablen zur späteren Modellanpassung verwendet werden, durch ein Optimierungsverfahren wählen zu lassen. Negative Seiteneffekte werden dadurch vermieden.

3.3.1 Univariate Filter

Univariate Filter zeichnen sich dadurch aus, dass zur Beurteilung der i-ten Variable lediglich x_i benutzt und alle übrigen x_j mit $i \neq j$ ignoriert werden. Einen Vorteil dieser Filter stellt die universelle Anwendbarkeit dar. Fehlende Werte, unterschiedliche Skalenniveaus oder große Datenvolumen stellen oft keine Probleme dar. Die meisten univariaten Filter benutzen lediglich ein Gütemaß, um jeder Variablen einen numerischen Wert, den Filterwert oder Score, zuzuordnen. Diese Filterwerte werden dann mit einem zuvor festgelegtem Grenzwert l verglichen, um zu einer binären Filterentscheidung zu gelangen. Alternativ wird häufig ein gewisser Prozentsatz f an Variablen festgesetzt, der den Filter passieren darf. Die schlechtesten $(1 - f)\%$ werden entsprechend herausgefiltert bzw. eliminiert.

In dieser Arbeit werden vier Typen von univariaten Filtern verwendet:

Varianz Zur Erklärung unterschiedlicher Überlebenszeiten ist eine gewisse Systematik und damit Varianz in den Variablen nötig, damit sich relevante Gene von sogenannten „Rauschgenen“ absetzen können. Entsprechend bedeutet eine hohe Varianz gleichzeitig eine hohe Relevanz.

Anpassung χ^2-verteilte Score-Statistik zur Anpassung des Cox Proportional Hazard-Modells (siehe Unterabschnitt 3.4.1). Bei einem degenerierten Modell kann diese Statistik irreführend sein, jedoch ist sie sehr günstig aus dem Modell zu extrahieren.

C-Index Konkordanzindex (siehe Unterabschnitt 3.2.1) zwischen numerischen Variablen und Überlebenszeit. Dieser Filter Kann als Korrelation zwischen Variable und Überlebenszeit interpretiert werden. Eine Überanpassung ist hier jedoch sehr wahrscheinlich.

Vorhersage Konkordanzindex zwischen Vorhersagen des Risikos eines univariaten Cox-Modells und den Überlebenszeiten, gemittelt über Evaluationen einer mehrfachen Aufteilung in Trainings- und Testmenge. Dies ist ein verhältnismäßig rechenintensiver Filter, da hier pro Variable mehrere Modelle angepasst und beurteilt werden müssen.

3.3.2 Minimale Redundanz, maximale Relevanz

Die Klasse der mRMR-Filter verwenden zur Beurteilung der Wichtigkeit einer Variablen einen Kompromiss aus deren Relevanz und Redundanz. Dieses Konzept ist ursprünglich für Klassifikation und lineare Regression entwickelt (Ding und Peng, 2005; Peng, Long und Ding, 2005), lässt sich jedoch einfach auf Überlebenszeitprobleme übertragen.

Das Verfahren ist iterativ und gierig. Im ersten Schritt wird jene Variable ausgewählt, welche hinsichtlich des gewählten Gütemaßes optimal ist. Im zweiten und jedem folgenden Schritt wird die Kovariable gewählt, welche einen Kompromiss aus hoher Güte bzw. Relevanz und gleichzeitiger geringer Redundanz besitzt. Die Redundanz bezieht sich dabei ausschließlich auf die bereits selektierten Variablen. Als Redundanzmaß wird häufig die Transinformation (*Mutual Information*) herangezogen, aber auch statistische Korrelationskoeffizienten sind hier möglich.

Für die konkrete Datenlage, wo das Hauptaugenmerk des Filterns auf den tausenden numerischen Variablen liegt, wurde als Relevanz der Konkordanzindex (siehe Unterabschnitt 3.2.1) gewählt. Als Redundanzmaß wird die Pearson-Korrelation

$$\mathrm{Cor}\,(x, y) := r_{xy} := \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

benutzt, indem in der i-ten Iteration für jede kandidierende Variable, welche noch nicht selektiert worden ist, die mittlere absolute Korrelation mit allen bereits selektieren Variablen berechnet wird. Die so errechnete Redundanz wird mit der Güte verrechnet, so dass sich ein einzelner Score ergibt. Dazu wird zumeist die Differenz oder der Quotient betrachtet, aber selbstverständlich sind alle erdenklichen Transformationen möglich und, wenn die Maße unterschiedlich skaliert sind, auch notwendig und sinnvoll. Da mit der Pearson-Korrelation und dem C-Index aber beide Maße auf dem Interval $[0, 1]$ definiert sind, spricht hier nichts gegen die Betrachtung des Quotienten. Seien also die Indizes der Kandidatenvariablen in Iteration i mit J_i und die Indizes der aktuell selektierten Variablen mit S_i gekennzeichnet, wobei $J_i \cap S_i = \varnothing$ und $J_i \cup S_i = \mathbb{N}_p$ gelte. Dann ist das $j^* \in J_i$ gesucht, welches unter Berücksichtigung des korrespondieren Konkordanzindexes C_j

$$j^* := \underset{j \in J_i}{\mathrm{argmax}} \left(\frac{C_j}{\frac{1}{n} \sum_{s \in S_i} |\mathrm{Cor}\,(x_j, x_s)|} \right)$$

erfüllt. Gerade die Berechnung der Korrelation gestaltet sich jedoch bei hochdimensionalen Daten schwierig. Ein vektorisierter Ansatz, welcher für `R`-Programme typisch ist, scheitert hier an dem benötigten Arbeitsspeicher der $(p \times p)$-Korrelationsmatrix, für iterative Ansätze wirft dagegen die Laufzeit in `R` Probleme auf. Verfügbare Pakete auf CRAN stellten sich ebenfalls als ungeeignet heraus, so dass das Paket `fmrmr` erstellt und auf GitHub veröffentlicht (Lang, 2014). Die volle $(p \times p)$-Korrelationsmatrix wird in dieser Implementierung zu keinem Zeitpunkt erzeugt, stattdessen wird lediglich ein $\mathbb{R}^p$-Vektor mit aggregierten Korrelationen iterativ aktualisiert. Durch die Verwendung von `C++` und der Matrix-Bibliothek `Armadillo` (Sanderson, 2010), angebunden über `RcppArmadillo` (Eddelbuettel und Sanderson, 2014), ist das Paket zudem von der Laufzeit hocheffizient.

3.3.3 Literaturfilter

Eine weitere Art des Filterns hochdimensionaler Überlebenszeitdaten stellen Filter dar, welche Informationen aus früheren Studien und Publikationen verwenden. Diese Literaturfilter definieren eine Menge von genetischen Kovariablen, welche sich entweder in ihrer biologischen Funktionalität interessant darstellen oder welche in vorherigen statistischen Analysen als relevant herausgestellt haben. Alle Gene, welche zur Definitionsmenge gehören, werden unabhängig von ihrer Ausprägung mit Wahrscheinlichkeit 1 ausgewählt, alle übrigen Gene werden verworfen. Für Lungenkrebsdaten ist solch ein Satz durch die „Kratz-Gene“ (Kratz u. a., 2012) gegeben, welcher 19 Kovariablen umfasst.

3.4 Regressionsmodelle

Das populärste Regressionsmodell der Überlebenszeitanalyse, das Cox Proportional Hazard-Modell (CoxPH), wird im anschließenden Unterabschnitt 3.4.1 vorgestellt. Da dieses Modell jedoch nicht auf hochdimensionalen Daten anwendbar ist, werden klassische Erweiterungen wie Strafterme und Boosting in den Unterabschnitten 3.4.2 und 3.4.3 vorgestellt. Als alternative Modellklasse für diese Daten werden in Abschnitt 3.5 Überlebenszeitbäume und deren Ensemble, Überlebenszeitwälder, eingeführt.

3.4.1 Cox Proportional Hazard-Modell

Für jede Beobachtung sei das Tripel (T_j, δ_j, Z_j) bestehend aus Ereigniszeit T_j, Nichtzensierungsindikator δ_j ($1 \triangleq$ unzensiert) und Kovariablenvektor $Z_j \in \mathbb{R}^p$ gegeben. Das von Cox (1972) vorgeschlagene Modell zur Modellierung der Hazardrate

$$\lambda(t \mid Z) = \lambda_0(t) \exp\left(\boldsymbol{\beta}^\top Z\right) \tag{3.1}$$

besteht aus zwei Termen, der Baseline-Hazard $\lambda_0(t)$, und den durch die Kovariablen erklärbaren Abweichungen von der Baseline als zweiter Term.

Die Schätzung $\hat{\boldsymbol{\beta}}$ des Koeffizientenvektors $\boldsymbol{\beta}$ erfolgt prinzipiell durch Maximum-Likelihood des zweiten Terms, daher wird entsprechend von „Partieller Likelihood“ gesprochen. Mit der Indexmenge der zum Zeitpunkt t unter Risiko stehenden Beobachtungen $R(t_i) =$

$\{j\colon t_j \geqslant t_i\}$ lässt sich die partielle Likelihood als

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{D} L_i = \prod_{i=1}^{D} \frac{\exp\left(\sum_{k=1}^{p} \beta_k Z_{ik}\right)}{\sum_{j \in R(t_i)} \exp\left(\sum_{k=1}^{p} \beta_k Z_{jk}\right)} \tag{3.2}$$

und die partielle Log-Likelihood als

$$\mathrm{LL}(\boldsymbol{\beta}) = \sum_{i=1}^{D} \ln\left(L_i\right) = \sum_{i=1}^{D} \sum_{k=1}^{p} \beta_k Z_{ik} - \sum_{i=1}^{D} \ln\left(\sum_{j \in R(t_i)} \exp\left(\sum_{k=1}^{p} \beta_k Z_{jk}\right)\right) \tag{3.3}$$

formulieren. Treten viele Bindungen (Ausfälle zum selben Zeitpunkt) auf, so ist die Partielle Likelihood

$$L_E(\boldsymbol{\beta}) = \prod_{i=1}^{D} \frac{\exp\left(\boldsymbol{\beta}^\top \left(\sum_{j \in D_i} Z_j\right)\right)}{\prod_{i=1}^{d_i} \left[\sum_{k \in R(t_i)} \exp\left(\boldsymbol{\beta}^\top Z_k\right) - \frac{j-1}{d_i} \sum_{k \in D_i} \exp\left(\boldsymbol{\beta}^\top Z_k\right)\right]}$$

von Efron (1977) zu präferieren. Dabei bezeichnet d_i die Ereignisanzahl zum Zeitpunkt t_i und D_i entspricht der Menge an Individuen, welche zum Zeitpunkt t_i das interessierende Ereignis erfahren.

Die Baseline-Hazardrate $\lambda_0(t)$ wird in einem anschließendem Schritt mittels Breslow-Schätzer

$$\hat{\lambda}_0(t) = \sum_{t_i \leqslant t} \left(\frac{d_i}{\sum_{j \in R(t_i)} \exp\left(\hat{\boldsymbol{\beta}}^\top Z_j\right)}\right)$$

bestimmt (Klein und Moeschberger, 2003).

Das CoxPH-Modell setzt proportionale Hazardraten voraus, also muss für zwei Stichprobenelemente mit Kovariablenvektoren Z_1 und Z_2 die Bedingung

$$\frac{\lambda(t \mid Z_1)}{\lambda(t \mid Z_2)} = \text{const.}$$

gelten. Dies ist gleichbedeutend mit zeitunabhängigen Kovariablen. Diese Annahme kann etwa durch Visualisierung der Residuen überprüft werden.

3.4.2 Regularisierung

Wenn die Anzahl der Variablen p die Anzahl der Beobachtungen n übersteigt, ist die Maximierung der Likelihood in (3.2) des CoxPH-Modells nicht mehr eindeutig lösbar. Eine klassische Lösung, sowohl für die $n \ll p$ Situation als auch für kollineare Regressoren in niedrigdimensionalen Datensituationen, besteht in der Einführung eines Strafterms zur Regularisierung. Für das CoxPH-Modell (3.1) ändert sich entsprechend das Optimierungsproblem der partiellen Log-Likelihood aus (3.3) mit der Einführung des Strafterms $\lambda \in \mathbb{R}_0^+$ zu

$$\underset{\beta}{\operatorname{argmax}} \left(\mathrm{LL}\left(\boldsymbol{\beta}\right) - \lambda \left((1-\alpha)\frac{1}{2} \left\| \boldsymbol{\beta}^2 \right\|_2 + \alpha \left\| \boldsymbol{\beta} \right\|_1 \right) \right).$$

Der Parameter $\alpha \in [0, 1]$ kontrolliert dabei die Art der Bestrafung. Für $\alpha = 0$, also ausschließlich quadratischer Bestrafung, wird von Ridge-Regression gesprochen (Hoerl und Kennard, 1970). Für $\alpha = 1$, nur absolute Bestrafung mittels L_1-Norm, wird von Lasso-Regression gesprochen (Tibshirani, 1996; Tibshirani, 1997). Für Werte $0 < \alpha < 1$, also einer Mischung beider Strafterme, wird die Bestrafung „Elastisches Netz“ (Zou und Hastie, 2005) genannt.

Eine nennenswerte Eigenschaft der Lasso-Regression ist deren implizite Variablenauswahl. Mit steigendem Strafterm λ werden zunehmend mehr Komponenten des Koeffizientenvektors $\boldsymbol{\beta}$ auf exakt Null geschätzt. In der Regel liefert die Lasso-Regression somit spärliche Lösungen, was insbesondere in der Analyse hochdimensionaler Daten eine wünschenswerte Eigenschaft für die weitere Modellanalyse darstellt. Die Ridge-Regression schrumpft dagegen die Komponenten von $\boldsymbol{\beta}$ lediglich sehr nahe, aber nicht exakt zur Null. Allerdings hat sich in mehreren Vergleichsstudien (siehe etwa Bøvelstad u. a. (2007) und Kammers u. a. (2011)) eine leichte Überlegenheit der Ridge-Regression hinsichtlich ihrer prädiktiven Güte gegenüber der Lasso-Regression herausgestellt. In der Praxis zeigt sich häufig, dass ein leichtes Einmischen des jeweils anderen Strafterms durch ein elastisches Netz das Modell verbessern kann, während gleichzeitig die jeweiligen Eigenschaften beibehalten werden. Je nachdem ob also Performanz oder Spärlichkeit präferiert wird, werden entsprechend α-Werte von $\alpha = 0.05$ bzw. $\alpha = 0.95$ gewählt.

3.4.3 Boosting

Die ursprüngliche Idee eines Boosting Algorithmus wurde von Schapire in „The strength of weak learnability“ (1990) entwickelt. In der Arbeit von Kearns und Valiant (1994)

wurde später das Konzept bewiesen: Es lässt sich stets ein kompetitiver Klassifikator als Ensemble aus schwachen Lernern, also Lernern, welche lediglich besser als zufällig sind, konstruieren. Daraus entwickelte sich schnell das populäre *Adaptive Boosting*, kurz AdaBoost (Freund und Schapire, 1995). AdaBoost gewichtet iterativ Beobachtungen neu und schiebt den Fokus so in Richtung der schwer vorherzusagenden Beobachtungen, um dort lokale Verbesserungen zu erreichen.

Basierend auf diesem Konzept entwickelten sich viele verschiedene auf spezielle Probleme zugeschnittene Anpassungen und Implementierungen. Für hochdimensionale Überlebenszeitdaten haben sich primär zwei Arten des Boostings durchgesetzt, das Modell-basierte Boosting bzw. *Gradient Boosting* sowie das Likelihood-basierte Boosting. Beide werden im Folgendem kurz vorgestellt.

Komponentenweises Gradient Boosting Zur Einfachheit bezeichnet y hier die Zielvariable, welche im Kontext der Überlebenszeitanalyse eigentlich für das Tupel (t, δ) steht. Das Gradient Boosting versucht, das Minimierungsproblem der Funktionsapproximation

$$F^* = \underset{F}{\operatorname{argmin}} \, \mathrm{E}_{y,\boldsymbol{X}} \left(L(y, F(\boldsymbol{X})) \right)$$

zu lösen (Friedman, 2001; Friedman, 2002), wobei $L : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$ eine beliebige aber differenzierbare Verlustfunktion bezeichnet. Dabei wird angenommen, $F(x)$ durch M Basislerner b_i mit Parametern $\boldsymbol{a} = (a_1, a_2, \ldots)^\top$ sowie zugehörigen reellen Gewichte β_i durch

$$F(\boldsymbol{X}) = \sum_{m=1}^{M} \beta_m b_m(\boldsymbol{X}, a_m) + \text{const.}$$

linear approximieren zu können. Zur Lösung wird mit einer initialen Schätzung für $m = 0$, etwa

$$F_0(\boldsymbol{X}) = \underset{c}{\operatorname{argmin}} \sum_{i=1}^{n} L\left(y_i, \boldsymbol{\beta}\right), \quad \text{für eine Konstante } c \in \mathbb{R},$$

gestartet. Anschließend werden die $m = 1, \ldots, M$ Schritte zur Schätzung von F_M wiederholt:

1. Bestimmte Pseudo-Residuen

$$\tilde{y}_{im} = -\left[\frac{\partial L\left(y_i, F(\boldsymbol{x}_i)\right)}{\partial F(\boldsymbol{x}_i)} \right]_{F(\boldsymbol{X}) = F_{m-1}(\boldsymbol{X})}$$

   als Richtung des steilsten Abstiegs der Verlustfunktion.

2. Modelliere auf Pseudo-Residuen den Basislerner durch Kleinste-Quadrate-Schätzung:

$$\boldsymbol{a}_m = \underset{\boldsymbol{a},\boldsymbol{\rho}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(\tilde{y}_{im} - \rho b(\boldsymbol{x}_i, \boldsymbol{a})\right)^2. \tag{3.4}$$

3. Löse das nun eindimensionale Minimierungsproblem

$$\boldsymbol{\beta}_m = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \sum_{i=1}^{N} L\left(y_i, F_{m-1}(\boldsymbol{x}_i) + \boldsymbol{\beta} b(\boldsymbol{x}_i, \boldsymbol{a}_m)\right).$$

4. Aktualisiere das Modell

$$F_m(\boldsymbol{x}) = F_{m-1}(\boldsymbol{x}) + \boldsymbol{\beta}_m b(\boldsymbol{x}, \boldsymbol{a}_m).$$

Für Überlebenszeitdaten müssen ein paar Anpassungen an dieses Verfahren vorgenommen werden, da durch die Zensierungen eine Schätzung von (3.4) unmöglich ist. Als Ausweg wird eine spezielle Verlustfunktion gewählt (Laan und Robins, 2003). Mit $\hat{G}$ als Schätzer der Überlebenszeitfunktion der Zensierungszeit C wird die empirische Risikofunktion nach Hothorn u. a. (2006) als

$$\hat{\mathrm{E}}\left(\tilde{y}, F(\boldsymbol{X}) \mid G\right) = \frac{1}{n} \sum_{i=1}^{n} L\left(\tilde{y}_i, F(\boldsymbol{x}_i)\right) \frac{\delta_i}{\hat{G}(T \mid \boldsymbol{x}_i)} \tag{3.5}$$

definiert. Dies entspricht einer üblichen Verlustfunktion, gewichtet mit der inversen Zensierungswahrscheinlichkeit – also dasselbe Konzept, welches auch beim Kaplan-Meier-Schätzer (Abschnitt 3.1.1) oder dem Brier-Score (Abschnitt 3.2.2) Anwendung findet. Der daraus resultierende Algorithmus inklusive seiner Parameter und Alternativen ist detailiert in Bühlmann und Hothorn (2007) beschrieben.

Likelihood-basiertes Boosting Eine andere Variation stellt das Boosting basierend auf der Likelihood dar. Es bietet die theoretische Möglichkeit, Kovariablen gruppiert durch $\mathcal{I}_{ml} \subseteq \{1, \ldots, p\}$ mit $l = 1, \ldots, q_m$ vordefinierten Mengen zu behandeln (Tutz und Binder, 2006). Somit kann auf einfache Art und Weise zusätzliches Wissen in den Boosting-Prozess einfließen. In jedem Iterationsschritt wird dazu die Verbesserung jeder Teilmenge gegenüber dem Modell des vorherigen Iterationsschritts untersucht, wobei alle Variablen der jeweiligen Teilmenge gezwungenermaßen betrachtet werden. Bezeichne $\eta_{i,m-1} = x^\top \beta_{m-1}$ den linearen Prädiktor aus dem vorherigem Schritt, λ einen

Strafparameter zur beliebigen Strafmatrix P_{ml} (etwa die Einheitsmatrix I_l), und $x_{i,\mathcal{I}_{ml}}$ den Kovariablenvektor für die i-te Beobachtung mit nur den Elementen, welche auch in $\mathcal{I}_{ml}$ vertreten sind. Dann kann das Optimierungsproblem für das CoxPH-Modell durch die Log-Likelihood

$$\mathrm{LL}(\gamma_{ml}) = \sum_{i=1}^{n} \delta_i \left(\eta_{i,m-1} + x_{i,\mathcal{I}_{ml}}^{\top} \gamma_{ml} - \log \left(\sum_{j=1}^{n} \mathbb{1}(t_j < t_i) \exp\left(\eta_{i,m-1} + x_{i,\mathcal{I}_{ml}}^{\top} \gamma_{ml} \right) \right) \right) - \lambda \gamma_{ml}^{\top} P_{ml} \gamma_{ml}$$

ausgedrückt werden (Binder und Schumacher, 2008a).

Dies erlaubt eine Sonderbehandlung von Kovariablen, wie es für klinische Daten in Überlebenszeitmodellen häufig gefordert wird. In der typischen Situation mit wenigen klinischen und tausenden genetischen Kovariablen kann so nicht nur lediglich ein Vorhersagemodell erzeugt und dessen genetische Kovariablen biologisch und medizinisch weiter gedeutet und analysiert werden. Stattdessen kann durch das Forcieren der klinischen Kovariablen implizit ein Modell mit genetischen Kovariablen mit einem sogenannten „additiven Wert“ gesucht werden, also nach Kovariablen welche im Vergleich mit den oft einfach zu erhebenden klinischen Variablen zusätzliche und relevante Information tragen. Dazu werden die Teilmengen als das kartesische Produkt aus den einelementigen Mengen von genetischen Kovariablen $\{\{1\}, \ldots, \{p\}\}$ mit der Menge der klinischen Kovariablen $\mathcal{I}_{\text{clin}}$ durch

$$\mathcal{I}_m = \{\{1\} \cup \mathcal{I}_{\text{clin}}, \ldots, \{p\} \cup \mathcal{I}_{\text{clin}}\}$$

gebildet, sodass bei jedem Optimierungsschritt die klinischen Variablen simultan Berücksichtigung finden. Das ganze bis hier beschriebene Verfahren ist in einen Newthon-Rhapson Algorithmus, siehe beispielsweise Lindstrom und Bates (1988), geschachtelt. Eine detailierte Beschreibung findet sich in Binder und Schumacher (2008a).

3.5 Überlebenszeitbäume und Überlebenszeitwälder

Bäume werden in der Klassifikation und Regressionsanalyse genutzt, um den Kovariablenraum $\mathcal{X}$ rekursiv zu partitionieren (Breiman u. a., 1984; Breiman, 1996). Dabei wird für jede Kovariable $\boldsymbol{x}_i$ jene Realisation gesucht, anhand der sich der Kovariablenraum in zwei möglichst „reine“ bzw. homogene Gruppen differenzieren lassen. Zur Klassifikation

wird häufig die Gini-Unreinheit

$$G = 1 - \sum_{j}^{k} p_j^2, \quad p_j \in [0, 1]$$

herangezogen, wobei p_j die relativen Häufigkeiten der insgesamt k Klassen in dem resultierenden Knoten bezeichnet. Der mittlere quadratische Fehler

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(y))^2$$

stellt ein populäres Analogon der Regressionsanalyse dar. Die binäre Aufteilung wird auf die Teilmenge der Beobachtungen in den entstandenen Zweigen bis zum erreichen eines Stoppkriteriums wiederholt angewendet, so dass die namensgebenede baumartige Struktur ensteht. Die Knoten, welche nicht weiter aufgeteilt werden, werden Terminalknoten oder Blätter genannt. Neue vorherzusagende Beobachtungen werden durch den Baum bis zu einem dieser Terminalknoten sortiert, welcher die finale Prädiktion beinhaltet. In der Klassifikation enthalten die Blätter ein Klassifikationslabel, in der Regressionsanalyse eine feste Zahl. Das resultierende Modell ist in Abbildung 3.1 exemplarisch dargestellt. Typische Stoppkriterien berücksichtigen beispielsweise die Anzahl der verbleibenden Beobachtungen im Knoten, die erzielte Reinheit oder die Komplexität, gemessen an der Gesamtzahl der Aufteilungen. Um einer möglichen Überanpassung entgegenzuwirken, werden Bäume nach ihrer Anpassung häufig gestutzt (*pruning*). Dabei werden iterativ die Blätter entfernt, welche in einer Kreuzvalidierung nicht entscheidend zur Verbesserung der Prädiktion beitragen können.

Bäume werden häufig zur Konstruktion von Ensembles als Zufallswälder (Breiman, 2001) genutzt. Die zugrundeliegende Idee ist dabei dem des Boosting nicht unähnlich: Aus vielen eher schwachen Lernverfahren kann ein starker und kompetitiver Prädiktor gebildet werden. Jedoch wird bei Wäldern im Gegensatz zum Boosting nicht ein Modell iterativ aktualisiert und verbessert. Stattdessen wird die endgültige Prädiktion über eine Aggregation der Prädiktion der einzelnen Basislerner benutzt. Bekannt ist hier vor allem das Mehrheitsvotum der einzelnen Bäume aus der Klassifikation, bei dem jene Klasse als Vorhersage gewählt wird, für die die Mehrheit der Bäume stimmen.

Für die Qualität des so konstruierten Prädiktors ist vor allem die Diversität der schwachen Lerner wichtig. Dazu werden die Basislerner bei Zufallswäldern auf Bootstrap-Stichproben (Ziehen mit Zurücklegen) und einer zufälligen Untermenge der Kovariablen gelernt. Auch

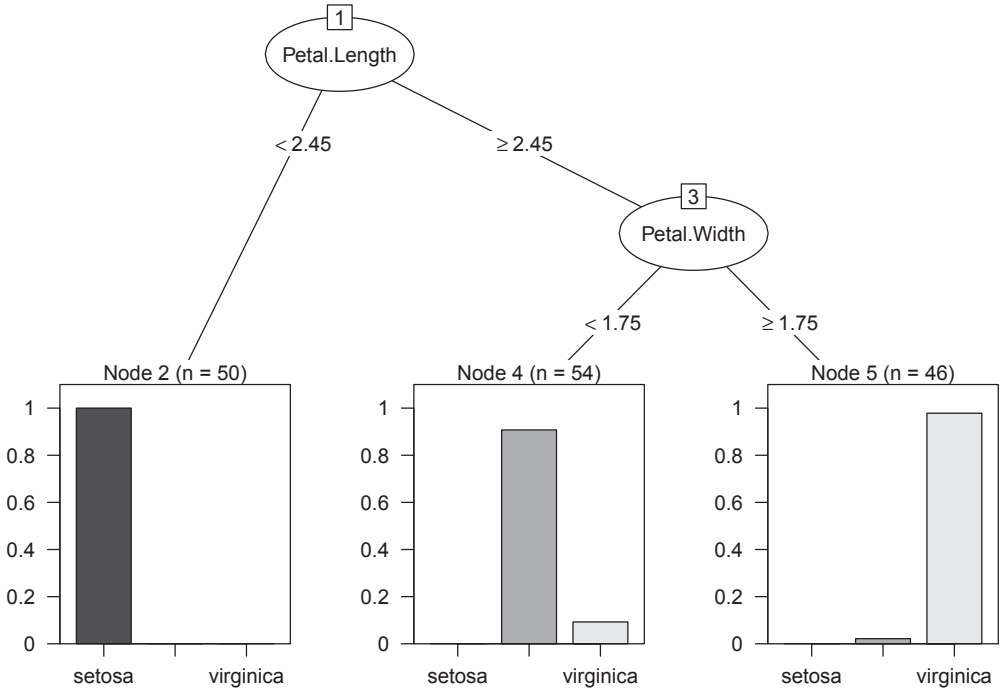

Abbildung 3.1: Beispiel eines Klassifikationsbaums zur Vorhersage der Spezies auf dem Iris-Datensatz (Anderson, 1935). Balkendiagramme visualisieren die relativen Häufigkeiten der drei Klassen in den Terminalknoten.

Überanpassung ist hier explizit bis zu einem gewissen Grad gewollt. Daher werden die einzelnen Bäume über Adaption des Stoppkriteriums verhältnismäßig „tief“ gelernt und auf ein anschließendes Stutzen der Bäume wird verzichtet, was einer hohen Spezialisierung gleichkommt. Dies hat den Effekt, dass auch für Regionen mit wenig Beobachtungen spezielle Bäume bzw. Zweige existieren und damit ebenfalls differenzierte Vorhersagen möglich sind. Anstatt also ein Modell zu finden, welches versucht alle Beobachtungen im Mittel zu treffen, werden viele „Experten“ demokratisch befragt.

Eine Adaption der Bäume und Wälder für Überlebenszeitdaten ist durch den Austausch einiger Komponenten problemlos möglich. Für das Maß zur Beurteilung der Reinheit wird zumeist der Logrank-Test (siehe Abschnitt 3.1.2) herangezogen. In den Blättern (Terminalknoten) des Baums befinden sich bei Überlebenszeitbäumen Beobachtungen ähnlichen Risikos, repräsentiert durch die zeitabhängige Hazardrate $\lambda(t)$ (siehe Abschnitt 3.1.2). Zur Beurteilung der Vorhersagen eines Baums kann aus dem Ensemble der kumulativen Hazardraten, geschätzt durch den Nelson-Aalen-Schätzer, die Mortalität geschätzt

werden. Diese ermöglicht ein Ranking der Beobachtungen, welches wiederum durch den Konkordanzindex beurteilt werden kann. Das Gesamtkonzept für Überlebenszeitbäume wird in LeBlanc und Crowley (1993) aufgezeigt, die Details der Erweiterung zu Wäldern sind in Hothorn u. a. (2004) und Ishwaran u. a. (2008) dargestellt.

4 Optimierung zur Modellwahl

Fast alle statistischen Lernverfahren sind auf eine gewisse Weise parametrisiert und teilweise sehr sensibel gegenüber Änderungen ihrer Parameter. Beispielsweise hat die Wahl der Kernfunktion einer Stützvektormaschine (SVM) einen gravierenden Einfluss, aber auch die Parameter eines speziellen Kernels sind sorgfältig zu wählen. Das finden einer „richtigen“ Parametrisierung stellt in der Praxis aber ein großes Problem dar, denn für die richtige Wahl ist zumeist ein Expertenwissen erforderlich ist, welches nur aus jahrelanger Praxiserfahrung gewonnen werden kann. Denn welche Parametrisierung für welches Modell auf welchen Daten zu wählen ist, bleibt zumindest teilweise Intuition. Wird die Vielzahl der Methoden und ihrer Implementierungen berücksichtigt, welche dank der Open-Source Bewegung der letzten Jahrzehnte jetzt zur freien und einfachen Verfügung stehen, wird schnell klar, dass automatisierte Lösungen zur Modellwahl und Parametrisierung wünschenswert sind.

Auf dem Weg zu solch einem Automatismus sind jedoch zunächst einige Vereinbarungen zu treffen. So muss ein Maß zur Beurteilung eines Modells bestimmt werden. Dazu kann prinzipiell jedes Gütemaß verwendet werden, jedoch sollte dieses aus einem Resampling-Ansatz bestimmt werden, um eine Überanpassung zu verhindern (Bischl u. a., 2012b). Zusammen mit einer maschinenlesbaren Beschreibung des Raums der Hyperparameter der Algorithmen ist dann ein allgemeines Optimierungsproblem definiert. Durch einen kleinen Kniff unterscheidet sich die Wahl des Modells, etwa ob ein Überlebenszeitwald oder ein penalisiertes CoxPH-Regressionsmodell angepasst werden soll, nicht von der Wahl eines beliebigen anderen nominalen Parameters. Dies wird in Abschnitt 4.3 erläutert. Zunächst wird jedoch das Problem der Algorithmenkonfiguration in Abschnitt 4.1 formal beschrieben. Anschließend wird in Abschnitt 4.2 die gewählte Optimierungsstrategie, die Modell-basierte Optimierung, vorgestellt. Abschnitt 4.4 schließt das Kapitel mit Informationen über die Implementierung ab.

4.1 Algorithmenkonfiguration und Hyperparameteroptimierung

Die Algorithmenkonfiguration hat das effiziente Finden einer möglichst guten Einstellung oder Konfiguration $\boldsymbol{\theta}$ in dem Raum der zugelassenen Einstellungen bzw. Hyperparameter Θ eines Algorithmus $\mathcal{A}$ zum Ziel. Dabei wird die Konfiguration $\boldsymbol{\theta}$ auf einer Probleminstanz $\omega \in \Omega$, etwa einer Resampling-Stichprobe, mittels einer Kostenfunktion $c_{\mathcal{A}} : \Omega \times \Theta \to \mathbb{R}$ beurteilt. Dies ist im Falle der Konfiguration von Maschinellen Lernverfahren in der Regel ein Gütemaß wie beispielsweise die Fehlklassifikationsrate oder etwa der Konkordanzindex.

Wir unterstellen Ω eine Wahrscheinlichkeitsverteilung $\mathcal{P}$ und definieren eine passende Zufallsvariable $\mathcal{I} \sim \mathcal{P}$. Da auch der Algorithmus $\mathcal{A}$ eine stochastische Komponente beinhalten kann, wird gegebenenfalls eine kombinierte Zufallsvariable $c_{\mathcal{A}}(\mathcal{I}, \boldsymbol{\theta})$ beobachtet. Das Optimierungsproblem lässt sich damit als

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} \, E\left[c(\mathcal{I}, \boldsymbol{\theta})\right] \tag{4.1}$$

formulieren. In der klassischen Algorithmenkonfiguration werden die Instanzen häufig als heterogen angenommen und die geschickte Auswahl von Instanzen ist integraler Teil der Optimierungsmethode. Dies vereinfacht sich für die Hyperparameteroptimierung insofern, dass die Instanzen als Resultat eines Resamplings nicht beeinflussbar oder selektierbar sind. Eine weitere Betrachtung ist somit an dieser Stelle nicht nötig.

Für ein konkretes Beispiel einer Hyperparameteroptimierung wird nun jene Konfiguration eines regularisierten CoxPH-Modells (vergleiche Unterabschnitt 3.4.2) gesucht, welcher zu der besten Güte, hier maximalem Konkordanzindex (vergleiche Unterabschnitt 3.2.1), führt. Der Hyperparameterraum setzt sich dabei aus dem Regularisierungsparameter λ und dem Parameter α, welcher zwischen absoluter und quadratischer Bestrafung balanciert, zusammen. Die Bedeutung dieser beiden Parameter ist in Unterabschnitt 3.4.2 erläutert. Der so aufgespannte Raum ist zweidimensional, wobei er hier durch die Nebenbedingungen $\lambda > 0$ und $\alpha \in [0, 1]$ konkret als $\Theta := [0, \infty) \times [0, 1]$ angegeben werden kann. Jeder Lösungskandidat $\boldsymbol{\theta} \in \Theta$ besteht also aus einem Tupel (λ, α). Die Aufgabe des Optimierers ist es nun, jenes $\boldsymbol{\theta}^*$ zu finden, welches zu einem möglichst hohen Konkordanzindex führt. Da der Konkordanzindex auf unabhängigen Testdaten bestimmt werden muss, handelt es sich unabhängig von dem regularisierten Modell um ein stochastisches Problem. In einer zehnfachen Kreuzvalidierung werden entsprechend zehn Probleminstanzen ω_i, $i = 1, \ldots, 10$, generiert und zur Reduktion des Einflusses besonders einfach oder schwer

zu lernender Instanzen wird das arithmetische Mittel der Konkordanzindizes dieser zehn Probleminstanzen gebildet. Als einfachen Optimierer betrachten wir für dieses Beispiel die Gittersuche. Wir gestatten dem Optimierer 100 Punktauswertungen, bevor er einen finalen Konfigurationsvorschlag $\boldsymbol{\theta}^*$ liefern muss. Zunächst muss dazu aber der Hyperparameterraum eingeschränkt werden, da eine Gittersuche auf dem Interval $[0, \infty)$ unmöglich ist. Solche Schranken sollten nach Möglichkeit konservativ gewählt werden, um interessante Bereiche vollkommen überdecken zu können. Hier wählen wir als Grenze $\lambda \leqslant 1000$ und legen entsprechend ein gleichmäßiges 10×10 Gitter in den Raum $[0, 1000] \times [0, 1]$. Da alle Punkte zehnfach kreuzvalidiert werden, verbraucht die Gittersuche insgesamt ein Budget von 1000 Modellevaluationen. Geschachteltes Resampling ist bei der Gittersuche nicht notwendig, da es sich hier nicht um ein adaptives Verfahren handelt. Der Punkt des Gitters mit maximalem gemittelten Konkordanzindex entspricht dem finalen Optimierungsvorschlag $\boldsymbol{\theta}^*$. Dieser kann in einem zweiten Schritt auf unabhängigen Testdaten evaluiert und mit den Ergebnissen anderer Optimierer verglichen werden.

Das gegebene Beispiel deckt ein großes Problem der Hyperparameteroptimierung nicht ab, welches durch die Struktur des Parameterraums Θ auftritt. Besitzt ein Algorithmus insgesamt l Parameter, so setzt sich Θ wie im Beispiel als Kreuzprodukt dieser zusammen, um so einen l-dimensionalen Raum aufzuspannen. Jedoch sind die Parameter häufig gemischt skaliert: Während einige Parameter reelle Zahlen annehmen können, sind für andere nur Ganzzahlen zulässig, wiederum andere besitzen dagegen ordinales oder nominales Skalenniveau. Zusätzlich können hierarchische Strukturen vorliegen: So sind manche Parameter nur dann zulässig, wenn für einen anderen Parameter gewisse Bedingungen gelten. Als Beispiel auf Ebene der Hyperparameteroptimierung kann erneut die Wahl des Kernels einer Stützvektormaschine herangezogen werden: Ein linearer Kernel besitzt andere Hyperparameter als ein Kernel mit radialer Basisfunktion und spannt daher einen anderen Hyperparameterraum auf. Solche Parameter werden als abhängig oder untergeordnet bezeichnet und treten insbesondere dann auf, wenn die Wahl des zugrundeliegenden statistischen Lernverfahrens ebenfalls als Hyperparameter eines Algorithmus aufgefasst wird, was in Abschnitt 4.3 genauer erläutert wird.

Im Folgenden wird die hier beschriebene Optimierung eines allgemeinen Algorithmus synonym auch als „Tuning“ bezeichnet. Gemeint ist dabei stets eine systematische Änderung der Hyperparameter mit dem Ziel, die erwarteten Kosten zu minimieren. Dabei ist zu beachten, dass Tuning stets auf einem Resampling der eigentlichen Trainingsdaten durchgeführt wird, die Auswertung also ein geschachteltes Resampling benötigt. Der Fehler auf den Trainingsmengen des inneren Resamplings wird dabei als Trainingsfehler

bezeichnet, der Fehler auf den Testmengen des inneren Resamplings, also noch auf der Trainingsmenge des äußeren Resamplings, wird dagegen als Optimierungsfehler definiert. Werden die gefundenen Modelle auf den Testmengen des äußeren Resamplings evaluiert, ist wie gewohnt vom Testfehler die Rede. Im anschließenden Abschnitt wird ein modernes Tuning verfahren vorgestellt, der generelle Ablauf des Tunings bleibt jedoch identisch.

4.2 Modell-basierte Optimierung

Ein Ansatz zur Optimierung des Problems (4.1) bietet die *Efficient Global Optimization* (EGO) (Jones, Schonlau und Welch, 1998) bzw. deren Verallgemeinerung, welche als Modell-basierte Optimierung (MBO) bezeichnet wird. MBO bedient sich eines Regressionsmodells, um die Güte $y := c_{\mathcal{A}}(\mathcal{I}, \boldsymbol{\theta})$ als Regressand den Hyperparametern $\boldsymbol{\theta}$ als Regressoren gegenüber zu stellen. Die Anzahl der Fehler, die ein Algorithmus begeht, wird also durch seine Parametereinstellungen (plus Rauschen) erklärt. Zum Vorschlagen neuer, interessanter Kandidatenkonfigurationen wird nun dieses Regressionsmodell – in diesem Kontext als Surrogatmodell bezeichnet – verwendet. Einerseits gibt die Modellierung der geschätzten Güte $\hat{y}$ Aufschluss über interessante Regionen. Zusätzlich birgt jedoch auch die Unsicherheit der Vorhersage, der lokale Varianzschätzer $\widehat{\mathrm{se}}\,(\boldsymbol{\theta})$ des Regressionsmodells, wertvolle Information. Denn in Regionen großer Unsicherheit, typischerweise schlecht explorierte Bereiche des Hyperparameterraums, steckt tendenziell mehr Optimierungspotenzial als in gut erforschten Bereichen. Das Verfahren läuft nach anfänglicher Initialisierung iterativ, wie in Algorithmus 1 beschrieben. Ein stark vereinfachtes Bei-

Algorithmus 1: Vereinfachter schematischer Ablauf der Modell-basierten Optimierung.

Initialisierung Werte b_{init} initiale Punkte aus.
wiederhole
  Passe Regressionsmodell auf allen bisher ausgewerteten Punkten an
  Bestimme den bzgl. eines Kriteriums interessantesten neuen Punkt $\boldsymbol{\theta}_{\text{infill}}$
  Werte $\boldsymbol{\theta}_{\text{infill}}$ aus
bis *Budget verbraucht*;
Bestimme finale Konfigurationen

spiel für das iterative Vorgehen der MBO ist in Abbildung 4.1 illustriert. Dort kommt ein Kriging-Modell und Expected Improvement zur Optimierung der Sinusfunktion als Zielfunktion zum Einsatz. MBO ist prinzipiell sehr generisch definiert und bietet viele austauschbare Komponenten, welche im weiteren diskutiert werden. Dabei liegt der Fokus

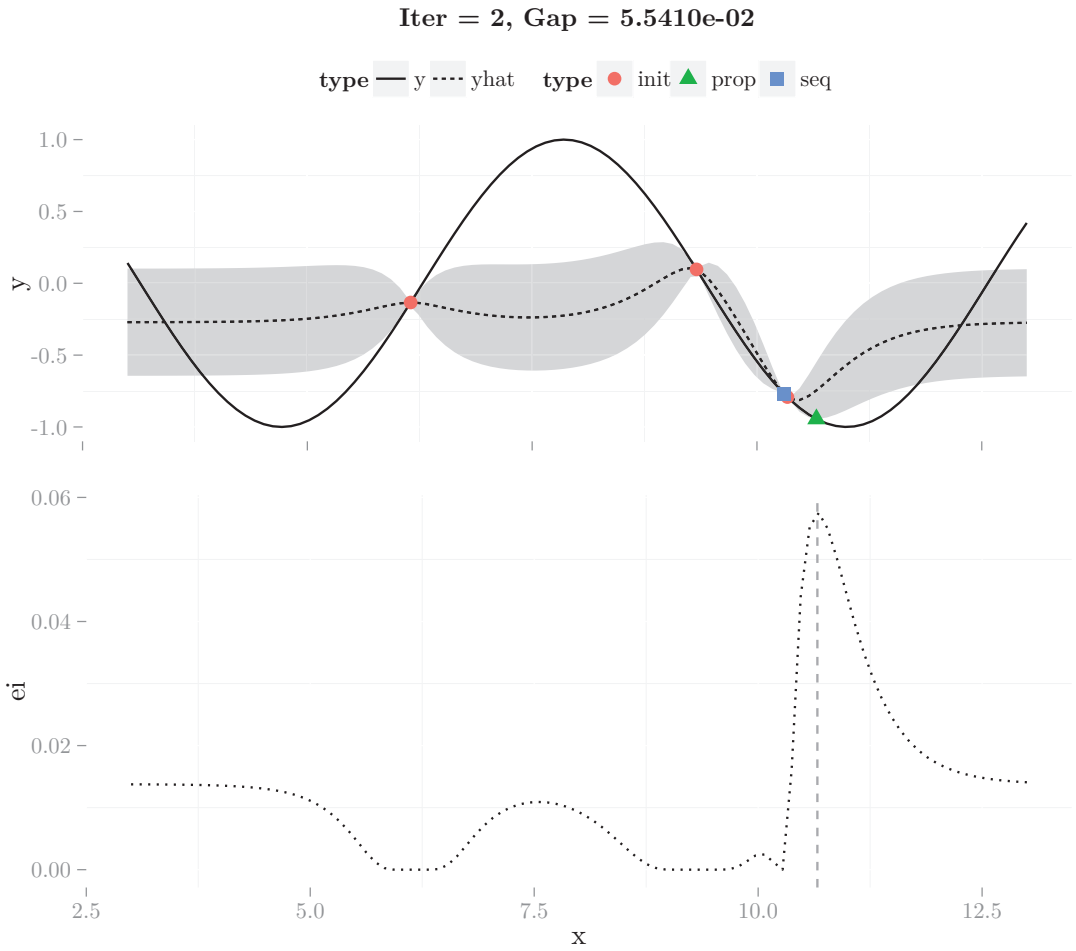

Abbildung 4.1: Beispielhafter Zustand in einer MBO-Iteration. Die durchgezogene Linie in der oberen Grafik entspricht der unbekannten, zu optimierenden Zielfunktion, hier $f(x) = \sin(x)$. Die roten Punkte wurden initial mittels LHS im Raum verteilt. Das blaue Quadrat symbolisiert einen zuvor (in der ersten Iteration) vorgeschlagenen und ebenfalls bereits evaluierten Punkt. Basierend auf diesen Punkten wurde ein Kriging-Modell angepasst (gestrichelte Linie). Die hervorgehobenen Bereiche entlang der gestrichelten Linie kennzeichnen die geschätzte lokale Unsicherheit. Das grüne Dreieck entspricht dem bezüglich Infill-Kriterium besten Punkt für diese zweite Iteration. Hier wurde das Expected Improvement gewählt, welches im unteren Teil der Grafik abgetragen ist.

klar auf den in der Auswertung verwendeten Methoden, ein vollständiger Überblick soll hier nicht erfolgen.

Regressionsmodell Die Wahl des Regressionsmodells richtet sich hauptsächlich nach der Struktur des Hyperparameterraums. Sind alle Hyperparameter numerisch, können theoretisch alle üblichen statistischen Regressionsmodelle verwendet werden, vorausgesetzt es existiert eine Möglichkeit zur Unsicherheitsschätzung. Hier haben sich insbesondere Kriging-Modelle (Krige, 1951; Rasmussen, 2004), oft auch als *Gaussian Process Regression* bezeichnet, profiliert, welche auch schon in der Originalarbeit zu EGO von Jones, Schonlau und Welch (1998) zum Einsatz kamen. Kriging-Modelle können vereinfacht als generelle lineare Modelle betrachtet werden, wobei die zu schätzende Zielfunktion als deterministisch angenommen wird. Der Fehlerterm des Modells entspringt dann einem stationären Gauß-Prozess, wobei die Kovarianz zweier Punkte als eine Funktion ihres Abstandes formuliert werden kann. Diese Modelle liefern einen gut interpretierbaren Schätzer des lokalen Vorhersagefehlers, da dieser einer a-posteriori Unsicherheit entspricht und proportional zur Distanz des nächsten evaluierten Punktes ist.

Für gemischt skalierte Parameterräume ist die Wahl des Regressionsmodells jedoch stark eingeschränkt. Zwar existieren einige vielversprechende Ansätze, etwa die *Bayesian Treed Gaussian Process Models* (Gramacy und Lee, 2008), um Kriging für faktorielle Parameterräume zu erweitern, jedoch bedarf es hier noch umfangreicher Untersuchungen. Eine vorgeschlagene Alternative stellen Zufallsregressionswälder dar (Hutter, Hoos und Leyton-Brown, 2011; Thornton u. a., 2012), welche zusammen mit den Überlebenszeitwäldern in Abschnitt 3.5 vorgestellt wurden. Regressionswälder besitzen einige herausragende Eigenschaften für das vorliegende Modellierungsproblem. Zum einen arbeiten sie per Konstruktion problemlos mit diskreten als auch numerischen Variablen. Darüber hinaus sind sie robust gegenüber Ausreißern in den Regressoren bzw. sind tolerant gegenüber stark isolierten Punkten, was gerade in den ersten Schritten der Optimierung häufig auftritt. Solche Punkte werden einfach ignoriert oder gegebenenfalls mit zusätzlichen Zweigen in den einzelnen Bäumen bedacht, die Vorhersage entfernter Punktmengen leidet jedoch nicht darunter. Dieses Verhalten kann sich darüber hinaus als Imputationsstrategie zunutze gemacht werden – fehlende Werte werden einfach durch sehr extreme Werte ersetzt. Das hat zur Folge, dass diese Werte, falls für die Vorhersagegüte nötig, durch die Bäume anders behandelt werden können als reguläre Beobachtungen. Fehlende Werte stellen die Mehrheit anderer Regressionsmodelle dagegen vor große Probleme, da hier derart einfache Imputationen nicht verfügbar sind.

Die Schätzung der Varianz der Vorhersage gestaltet sich bei Bäumen dagegen schwierig. Für Wälder existieren Bootstrap-Schätzer, bei denen mehrere Wälder auf Bootstrap-Stichproben angepasst werden, um sich aus den jeweiligen Vorhersagen ein Streuungsmaß zu konstruieren. Alternativ können Jackknife-Schätzer benutzt werden, bei denen die Varianz auf jenen Beobachtungen bestimmt wird, welche für die Anpassung eines einzelnen Baumes nicht berücksichtigt wurden (sogenannte *out-of-bag samples*). Beide Varianten werden detailiert in Sexton und Laake (2009) und Wager, Hastie und Efron (2014) untersucht. Für diese Arbeit wurde der übliche Bootstrap-Schätzer gewählt. Bei weniger teuren Problemen, wo die benötigte Rechenzeit zur Anpassung eines Waldes im Verhältnis zu einer Punktauswertung ins Gewicht fällt, wird besser auf den Jackknife-Schätzer zurückgegriffen, da dieser lediglich einen einzigen Wald benötigt. Beide Varianten haben jedoch einen gemeinsamen Nachteil, denn anders als beim Kriging entspricht die berechnete Vorhersagevarianz nicht im engeren Sinne einer Unsicherheit. Regressionsbäume und somit auch die Wälder vernachlässigen Regionen mit geringer Datenmasse in der Partitionierung des Raums. Das hat zur Folge, dass – unabhängig von der Entfernung zum nächsten gelernten Punkt – eine konstante Vorhersage für solche Regionen erfolgt. Ein Zusammenhang zwischen Varianz und Distanz liegt also nicht unmittelbar vor. Gerade deswegen ist ein raumfüllendes Design zur Initialisierung des Verfahrens essentiell, denn nur wenn alle Regionen zumindest sporadisch besucht worden sind, können sich die Vorhersagen und somit auch deren Varianzen sinnvoll voneinander unterscheiden.

Infill-Kriterium Das Infill-Kriterium bezeichnet das Kriterium zum Vorschlagen neuer Punkte bzw. Kandidaten zur potenziellen Auswertung. Als Grundlage dient stets die Prädiktion aus dem Surrogatmodell sowie die lokale Unsicherheit an der jeweils betrachteten Stelle. Ein sehr populäres Kriterium stellt dabei die Erwartete Verbesserung (*Expected Improvement*, EI) dar. Die Verbesserung I wird nach Jones, Schonlau und Welch (1998) an der Stelle $\boldsymbol{\theta} \in \Theta$ durch

$$I(\boldsymbol{\theta}) = \max\left(y_{\min} - y(\boldsymbol{\theta}), 0\right)$$

beschrieben, wobei $y_{\min}$ der derzeit beste bisher beobachtete Funktionswert und $y(\boldsymbol{\theta}) := c_{\mathcal{A}}(I, \boldsymbol{\theta})$ die in Abschnitt 4.1 beschriebene Zufallsvariable des Funktionswertes an der Stelle $\boldsymbol{\theta}$ bezeichnet. Unter der Voraussetzung, dass $\hat{y} \sim \mathcal{N}(\hat{y}(\boldsymbol{\theta}), \hat{s}^2(\boldsymbol{\theta}))$ normalverteilt sei, wie es im Kriging-Modell gilt, kann die Erwartete Verbesserung an jedem Punkt $\boldsymbol{\theta} \in \Theta$

durch Betrachtung des Erwartungswertes nach Jones, Schonlau und Welch (1998) als

$$\mathrm{E}\left[I(\boldsymbol{\theta})\right] = (y_{\min} - \hat{y}(\boldsymbol{\theta}))\,\Phi\left(\frac{y_{\min} - \hat{y}(\boldsymbol{\theta})}{s}\right) + s\phi\left(\frac{y_{\min} - \hat{y}(\boldsymbol{\theta})}{s}\right)$$

ausgedrückt werden. Dabei bezeichnet $\Phi(\cdot)$ die Verteilungsfunktion der Standardnormalverteilung und $\phi(\cdot)$ deren Dichte. Der vorgeschlagene Punkt ergibt sich aus Maximierung des Expected Improvements, also $\boldsymbol{\theta}^* = \operatorname{argmax}_{\boldsymbol{\theta}} \mathrm{E}\left(I(\boldsymbol{\theta})\right)$. Die Verwendung eines Kriging-Modells schließt sich für das gegebene Problem der Hyperparameteroptimierung jedoch aus, da es sich einerseits um ein stochastisches Problem handelt und andererseits das Kriging nicht für nominale Variablen geeignet ist. Die Rechtfertigung der Erwarteten Verbesserung als Maß ist somit ebenfalls schwierig. Das alternative Kriterium *Lower Confidence Bounds* (LCB) (Snoek, Larochelle und Adams, 2012) ist mit

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_{\boldsymbol{\theta}\in\Theta} \mathrm{LCB}(\boldsymbol{\theta}) = \operatorname*{argmin}_{\boldsymbol{\theta}\in\Theta} \left(\hat{y}(\boldsymbol{\theta}) - \lambda_{\mathrm{LCB}}\hat{s}(\boldsymbol{\theta})\right) \tag{4.2}$$

dagegen – einfach als untere Schranke des Konfidenzbandes – verteilungsfrei motiviert und kann als modellunabhängige Heuristik aufgefasst werden. Der Parameter $\lambda_{\mathrm{LCB}} \in \mathbb{R}^+$ dient dabei der Gewichtung zwischen Mittelwert und Varianz und ermöglicht die Balancierung zwischen Exploitation und Exploration. Für kleines λ_{LCB} werden gute Regionen weiter ausgeschöpft, große Werte von λ_{LCB} präferieren eine Exploration des Raums. Darüber hinaus existiert ein einfacher aber vernünftiger Ansatz zum gleichzeitigen Vorschlagen mehrerer Punkte: Sollen q Punkte gleichzeitig vorgeschlagen werden, können λ_{LCB}-Werte zufällig aus einer Exponentialverteilung gezogen werden (Bischl u. a., 2014c). Die so vorgeschlagenen Punkte können entsprechend parallel evaluiert werden.

Infill Optimierer Um den optimalen Wert des Surrogatmodells zu finden, per Definition hier also das Minimum, muss bei vielen Regressionsmodellen erneut ein Optimierer eingesetzt werden. Hier sind erneut viele Herangehensweisen denkbar, die später in der Auswertung benutzte ist die Fokussuche (*Focus Search*). Dieses Verfahren ist noch nicht publiziert, sondern lediglich durch das Paket `mlrMBO` (Bischl u. a., 2014b) von uns bereit gestellt. Ein großer Vorteil der Fokussuche stellt die Flexibilität dar, da sie problemlos auf gemischt-skalierten Parameterräumen arbeitet. Zudem ist sie als verhältnismäßig einfaches Verfahren robust gegenüber technischen Problemen im Surrogatmodell: Sollte ein Punkt oder ganze Mengen von Punkten in gewissen Regionen nicht vorhersagbar sein, ist dennoch ein fehlerfreier Ablauf der Optimierung gewährleistet. Mitunter ist die

Fokussuche auch durch die Eigenschaften von `R` motiviert. Die in `R` typische Vektorisierung hat den Effekt, dass die gleichzeitige Vorhersage vieler Punkte durch das Surrogatmodell wesentlich günstiger ist als iterativ Vorhersagen für einzelne Punkte zu generieren.

Bei der Fokussuche wird der Parameterraum Θ zunächst durch zufälliges *Latin-Hypercube-Sampling* (LHS) (Stein, 1987) mit n_l Punkten grob aber raumfüllend besucht und der resultierende beste beobachtete Punkt $\boldsymbol{\theta}^*$ bestimmt. In einem nächsten Schritt wird auf die Region um $\boldsymbol{\theta}^*$ fokussiert, indem der Parameterraum zu $\boldsymbol{\theta}^*$ hin geschrumpft wird: Der erlaubte Bereich numerischer Parameter wird halbiert, wobei der Mittelpunkt, falls möglich, auf $\boldsymbol{\theta}^*$ liegt. Sollte der Punkt am Rand des erlaubten Bereichs liegen, kann die Dimension im Extremfall auch geviertelt werden. Für diskrete Parameter wird zufällig eine mögliche Ausprägung eliminiert, die nicht der entsprechenden Ausprägung in $\boldsymbol{\theta}^*$ entspricht. Anschließend werden erneut mittels LHS n_l Punkte in den jetzt verkleinerten Raum gelegt und abermals fokussiert, wobei die Anzahl dieser Wiederholungen als n_f bezeichnet sei. Das gesamte Vorgehen wird mehrmals wiederholt und der beste Punkt aus allen Neustarts ermittelt. Diese n_r Neustarts sollen verhindern, dass eine einzelne voreilige Fokussierung im ersten Schritt zu viel Einfluss auf das Endresultat hat. Der Pseudo-Code zur Fokussuche ist in Algorithmus 2 zusammengefasst.

Finaler Modellvorschlag Jede Punktauswertung wird in einer Kreuzvalidierung mehrmals durchgeführt, um die zufälligen Schwankungen zu mindern. Der finale Vorschlag ermittelt sich hier entsprechend als jene Konfiguration, welche zu dem besten gemittelten beobachteten Funktionswert geführt hat.

4.3 Modellselektion als Optimierungsproblem

In diesem Abschnitt wird der Schritt von der Optimierung zur Modellselektion beschrieben. Eine Zielfunktion zur Konfiguration eines einzelnen Algorithmus wurde bereits in Abschnitt 4.1 definiert. Zur Erweiterung für mehrere Modelltypen wird der Parameterraum um eine Variable erweitert. Diese faktorielle Variable spezifiziert den Modelltyp, während alle Parameter eines jeweiligen Modelltyps auf eine einzige mögliche Ausprägung bedingen. Eine Veranschaulichung dazu findet sich in Abbildung 4.2. Zeigt der neue Hyperparameter `selected.learner` hier den Überlebenszeitwald als aktives Modell an, macht die Einstellung von `sigma` keinen Sinn. In diesem Fall wird `sigma` als inaktiv oder deaktiviert bezeichnet. Dieser Mechanismus ist hauptsächlich für den hierarchischen

Algorithmus 2: Schematischer Ablauf der Fokussuche.

Definiere $\mathbb{D}$ als beschränkten p-dimensionalen Hyperparameterraum mit

$$\bigtimes_{i=1}^{p} d_i, \quad d_i = \begin{cases} [l_i, u_i], & d_i \text{ numerisch} \\ \{f_1, \ldots, f_m\}, & d_i \text{ nominal} \end{cases}$$

Initialisiere $\hat{y}_{\text{best}} := \infty$
für alle $k = 1, \ldots, n_r$ *Neustarts* **tue**
  für alle $j = i, \ldots, n_f$ *Fokusierungen* **tue**
    Sample n_l Punkte $\boldsymbol{\theta}_{\text{prop}} = \left\{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_{n_f}\right\}$ mittels LHS im Raum $\times_{i=1}^{p} d_i$
    $\boldsymbol{\theta}^* := \text{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\theta}_{\text{prop}}} \hat{y}(\boldsymbol{\theta})$
    für alle $i = 1, \ldots, p$ *Dimensionen* **tue**
      wenn d_i *numerisch* **dann**
        $d_i := \left[\max(l_i, \boldsymbol{\theta}_i^* - \frac{|l_i - u_i|}{4}), \min(u_i, \boldsymbol{\theta}_i^* + \frac{|l_i - u_i|}{4})\right]$
      sonst
        wenn $|d_i| \geqslant 2$ **dann**
          $d_i := d_i \setminus \omega, \omega \sim \mathcal{U}(d_i \setminus \boldsymbol{\theta}_i^*)$ diskret gleichverteilt
  wenn $\hat{y}_{\text{best}} > \hat{y}(\boldsymbol{\theta}^*)$ **dann**
    $\boldsymbol{\theta}_{\text{best}} := \boldsymbol{\theta}^*$
Ausgabe : $(\boldsymbol{\theta}_{\text{best}}, \hat{y}(\boldsymbol{\theta}_{\text{best}}))$

Parameterraum verantwortlich, was wiederum eine Imputationsstrategie für das Surrogatmodell erforderlich macht. Um Konfusionen mit Meta-Lernen zu vermeiden haben wir die Implementierung in Analogie zum Bauteil der Elektrotechnik „Multiplexer“ genannt.

Die Verwendung des Multiplexers hat gegenüber dem separaten Tunen jedes einzelnen Modells einen wichtigen Vorteil: Oft ist schon nach ein paar Iterationen abzusehen, dass manche Modellklassen auf dem aktuellen Datensatz stets unterlegen oder überlegen sind. In dem Setup, in dem alle separat getunte Lerner miteinander verglichen werden, wird dieser Umstand jedoch ignoriert und viel Laufzeit darauf verschwendet, „schlechte“ Modelle ein wenig besser zu machen. Stattdessen kann diese Laufzeit aber auch verwendet werden, den Raum bedingt auf die „guten“ Modell weiter zu explorieren bzw. das gewonnene Budget zur Optimierung der überlegenen Modelle zu verwenden. Genau das wird implizit durch die Anwendung eines Optimierers auf solch einen konstruierten Multiplexer erreicht. Ein Sprung zurück zu den unterlegenen Modellen ist dabei stets möglich und erfolgt, sobald die anderen Modelle gründlich exploriert sind und damit die lokale Unsicherheit verhältnismäßig klein ist.

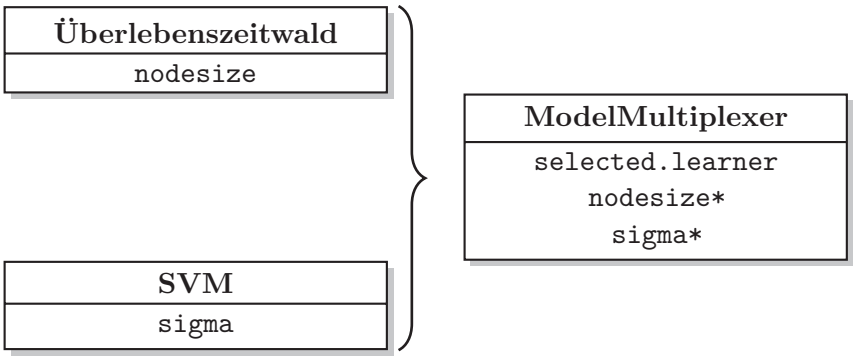

Abbildung 4.2: Beispiel eines Modell-Multiplexers. Fusion eines Überlebenszeitwaldes mit einer Stützvektormaschine zu einem kombinierten Lernverfahren. Die jeweiligen Hyperparameter `nodesize` und `sigma` werden zu abhängigen Parametern des kombinierten Lernverfahrens und bedingen auf die Ausprägung des neuen Hyperparameters `selected.learner`.

4.4 Implementierung

Da der Optimierer im Hyperparameterraum operiert, ist eine abstrakte Beschreibung des Raums unabdingbar. Das Paket `ParamHelpers` (Bischl u. a., 2014d) implementiert das notwendige Grundgerüst, um alle üblichen Parametertypen abzubilden. Dies beinhaltet Skalare und Vektoren aller Basis-Datentypen: *logical*, *integer*, *numeric*, und *discrete*. Zusätzlich existiert noch ein weiterer Typ *untyped* für Parameter, welche entweder mehr als einen atomaren Datentyp akzeptieren oder nicht-atomare Datentypen wie Funktionen erwarten. Da Ein- und Ausgaben der einzelnen Modellierungsschritte

(a) Aufteilung in Trainings- und Testdaten,

(b) Lernen des Modells auf Trainingsdaten,

(c) Vorhersagen des Modells auf Testdaten, und

(d) Evaluierung mittels Gütemaß

stets wohldefiniert sind, ist eine Abstraktion möglich und sinnvoll. Das Paket `mlr` (Bischl u. a., 2014a) bietet unter anderem genau solch eine Funktionalität und wurde entsprechend für die Überlebenszeitanalyse erweitert. Darüber hinaus bietet `mlr` die Möglichkeit, Lernverfahren durch Wrapper zu erweitern: Der Lern- und Vorhersageschritt kann in beliebige `R`-Funktionen „gewickelt“ werden. Auf diese Art und Weise sind der Multiplexer zur Fusionierung mehrerer Lernverfahren, die vorgeschalteten Variablenfilter der

Überlebenszeitmodelle, die Imputation fehlender Werte für das Surrogatmodell sowie das Tuning selbst implementiert und stellen somit konzeptionell nur Erweiterungen der Basis-Lernverfahren dar.

Für die Experimente in Lang u. a. (2015) wurden Überlebenszeitmodelle mit viel Aufwand in Funktionen gekapselt und so eine minimale Abstraktion als Vorarbeit für die Überlebenszeit-Erweiterung von `mlr` geschaffen. Jedoch fehlte zusätzliche Funktionalität, welche teilweise bereits in `mlr` vorhanden war und nachgebildet werden musste, teilweise jedoch auch neu geschaffen wurde und später in das Paket eingearbeitet wurde:

- Behandlung von neuen Faktorstufen in der Vorhersage, verursacht durch Resampling.
- Konvertierung der Eingabedaten. Viele Überlebenszeitmodelle arbeiten ausschließlich auf numerischen Matrizen, diskrete Variablen müssen zuvor durch zu 0/1-kodierten Variablen konvertiert werden.
- Stratifizierte Kreuzvalidierung. Für Überlebenszeitmodelle ist die Stratifizierung auf der Zensierungsvariable elementar. Denn falls keine Ereignisse im Testdatensatz vorhanden sind, ist eine Berechnung des Konkordanzindexes technisch unmöglich. Weiter verursachen leere Faktorstufen Probleme, was ebenfalls durch Stratifizierung vermieden werden kann.
- Abfangen von unerwarteten Programmabbrüchen und Behandlung durch Imputation ausgebliebener Modelle und Gütemaße.
- Parallelisierung auf verschiedenen Niveaus. Geschachtelte Parallelisierung durch `BatchJobs` (Bischl u. a., 2012a) und `parallelMap` (Bischl und Lang, 2014). Auf die Parallelisierung wird im anschließenden Kapitel 5 ausführlich eingegangen.

MBO selbst ist in das externe Paket `mlrMBO` (Bischl u. a., 2014b) ausgelagert, ist jedoch ebenfalls eng mit `ParamHelpers` und `mlr` verzahnt. Das Paket ist sehr modular aufgebaut, so dass alle in Abschnitt 4.2 vorgestellten Komponenten verändert und erweitert werden können. So kann beispielsweise jedes Regressionsverfahren, welches eine Varianzschätzung anbietet und als `mlr`-Lernverfahren verfügbar ist, als Surrogatmodell genutzt werden. Weiter finden sich im Paket Erweiterungen für multikriterielle Optimierung (Horn u. a., 2015) sowie Optimierung unter Berücksichtigung von Kosten. Eine Veröffentlichung auf CRAN steht hier noch aus.

5 Paralleles Rechnen in R

Seit einigen Jahren scheint das Mooresche Gesetz, dass sich die Komplexität integrierter Schaltkreise (und somit Rechenleistung) regelmässig verdoppelt, außer Kraft gesetzt. Die Hardwareentwicklung fokussiert sich nicht mehr primär auf die Schrumpfung von Transistoren und den damit verbundenen physikalischen Problemen, sondern setzt verstärkt auf Parallelität. Gleichzeitig haben sich die Möglichkeiten der automatischen Datenerfassung und Datenverarbeitung rasant entwickelt. Datensätze von Internet-Plattformen umfassen oft viele Millionen Beobachtungen ($n \gg p$, *big data*) und Hochdurchsatz-Technologien liefern Daten zu bis zu 500.000 Einzelnukleotidpolymorphismen (SNPs) gleichzeitig ($n \ll p$, *high-dimensional data*). Auf solchen Daten ist sogar bei Computerexperimenten moderaten Umfangs eine parallele Ausführung unverzichtbar.

In diesem Kapitel wird zunächst in Abschnitt 5.1 die Funktionsweise von Aufrufen trivial parallelisierbarer Funktionen erläutert. Folgend werden in Abschnitt 5.2 verschiedene `R`-Pakete und ihre Zielarchitekturen vorgestellt. Anschließend werden in den Abschnitten 5.3 und 5.4 selbst entwickelte Pakete zur Parallelisierung näher beleuchtet. Abschnitt 5.5 schließt dieses Kapitel mit einer Diskussion über Reproduzierbarkeit im parallelen Rechnen ab.

5.1 Map-Reduce Paradigma

`R` lässt sich (mit wenigen Ausnahmen) als funktionale Sprache auffassen. Das bedeutet unter anderem, dass `R` Möglichkeiten zum Kombinieren und Transformieren von Funktionen anbietet und Funktionen gleichberechtigt wie andere Datentypen behandelt werden. So lassen sich häufig auftretende Teilaspekte der Programmierung elegant und verständlich durch sogenannte Funktionen höherer Ordnung, also Funktionen, welche Funktionen als Argumente akzeptieren, lösen. Der wohl bekannteste Vertreter der Funktionen höherer Ordnung ist die *Map*-Funktion. Die übergebene Funktion wird dabei auf das erste Element

jeder der Eingaben $x_1, \ldots, x_k$ mit $x_i = (x_{i1}, \ldots, x_{in})$ angewendet, auf das zweite Element, ..., auf das n-te Element:

$$\text{map}\,(f, x_1, \ldots, x_k) = \begin{pmatrix} f(x_{11}, \ldots, x_{k1}) \\ \vdots \\ f(x_{1n}, \ldots, x_{kn}) \end{pmatrix}, \quad x_i \in \mathbb{I}^n, f : \mathbb{I}^k \to \mathbb{O}.$$

Dabei bezeichnet $\mathbb{I}$ einen beliebigen Körper auf dem die Funktion f operieren kann, $\mathbb{O}$ kann unspezifiziert bleiben. In `R` ist die Map-Operation als `mapply()` und `Map()` bzw. für den Speziallfall $k = 1$ als `lapply()` implementiert. Diese Funktionen sind nicht zuletzt aufgrund der inperformanten Schleifenausführung früherer `R` Versionen populär geworden und deren Verwendung ist bis heute *good practice*. Eine Map-Operation kann prinzipiell alle `for()`-Schleifen mit unabhängigen Iterationen ersetzen. Die Parallelisierung ist durch die Unabhängigkeit ebenfalls trivial.

Eine andere populäre Operation stellt das *Reduce* dar. Es ist als rekursiver Funktionsaufruf einer Funktion $f : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ mit Startwert x_0 definiert. Für $k = 3$ etwa ist die Reduce-Operation als

$$\text{reduce}\,(f, x_1, x_2, x_3) = f\,(f\,(f\,(x_0, x_1)\,, x_2)\,, x_3)$$

definiert. Beispielsweise könnte die Summation eines Vektors $x \in \mathbb{R}^k$ durch die Funktion $f(x, y) = x + y$ mit $x_0 = 0$ als Reduktion ausgedrückt werden. Ist die Komposition von f assoziativ und kommutativ, so ist eine Parallelisierung ebenfalls problemlos als mehrschrittiges Baumverfahren möglich: Die Eingabe wird dazu in k Teile partitioniert und die Reduktion auf jede der Partitionen parallel ausgewertet. Die resultierenden k Reduktionen können als neue Eingabe verwendet werden und erneut parallel ausgewertet werden, bis eine finale Reduktion das gewünschte Endergebnis liefert.

5.2 Architekturen und Pakete

Abhängig von der zugrundeliegenden Hardwarearchitektur sind verschiedene Parallelisierungsansätze und auch `R`-Pakete nötig.

Die einfachste (weil lokale) Parallelisierung nutzt auf einer einzelnen Maschine mehrere verfügbare CPUs und CPU-Kerne. Die Kommunikationskosten können typischerweise vernachlässigt werden, da auf dem selben Adressraum operiert werden kann und somit keine

Duplikation oder Kommunikation von möglicherweise voluminösen Ein- oder Ausgabedaten anfällt. Dies gilt auch für die Map-artigen Operationen `mclapply()` und `mcmapply()`, welche in dem Basispaket `parallel` angeboten werden. Durch die Ausnutzung von Forks ist eine Kopie der Daten auch hier erst bei schreibendem Zugriff nötig. Die offensichtliche Limitierung dieser Parallelisierung stellt die geringe Anzahl möglicher verfügbarer CPUs auf einem einzelnen System dar.

Der naheliegende nächste Schritt ist somit der Zusammenschluss mehrerer lokal vernetzter Rechner zu einem Rechnerverbund (Cluster). Kleinere Cluster mit bis zu ein paar Dutzend Maschinen sind häufig dezentral organisiert: Benutzer können sich auf mindestens einer Maschine einloggen und über standardisierte Kommunikationsschnittstellen wie *Sockets* oder dem *Message Passing Interface* (MPI) (Gropp, Lusk und Skjellum, 1999) Berechnungen auf anderen Maschinen des Verbunds anstossen. Map-artige Funktionen existieren ebenfalls im Basispaket `parallel` als Funktionen `parLapply()` bzw. `clusterMap()`. Eine populäre Alternative dazu stellt das `snowfall`-Paket (Knaus, 2013) dar. Die effiziente Auslastung derartiger Cluster ist jedoch in heterogenen Netzwerken, also Netzwerken, in denen sich die Ressourcen einzelner Knoten deutlich unterscheiden, schwierig und benötigt relativ viel manuelles Eingreifen zur optimalen Verteilung der einzelnen Jobs. Darüber hinaus stellt der Mehrbenutzerbetrieb ein Problem dar, da das System einfach und insbesondere unabsichtlich überbeansprucht werden kann. Gerade eine Überallokation von Arbeitsspeicher führt oft zum faktischen Ausfall des gesamten Rechenknotens, inklusive Abbruch aller Benutzerprozesse.

Um diese Probleme zu adressieren wird häufig ein Scheduler eingesetzt. Seine Aufgabe ist die gerechte und effiziente Aufteilung verfügbarer Ressourcen, welche von Nutzern durch Jobs angefordert werden. Die Jobs setzen sich in der Regel aus auszuführendem Shell-Kommando sowie der Spezifikation der Anzahl benötigter CPUs, der erwarteten maximale Laufzeit und dem maximalen Speicherbedarf zusammen. Diese werden dann in einem Scheduler-spezifischen Format mittels eines Kommandozeilentools an den Scheduler abgesetzt, welcher diese Anfragen einer Warteschlange hinzufügt. Die Reihenfolge der Abarbeitung wird durch das Rechenkontingent der Nutzer sowie der verfügbaren Ressourcen bestimmt. Als Besonderheit ist hier die Verzögerung und Asynchronität einzelner Jobs zu sehen. Dies verbietet insbesondere eine Kommunikation zwischen Jobs. Für solche Fälle muss dann stattdessen ein umfangreicher Job mit vielen parallelen CPUs angefragt werden. Andererseits erlaubt die Aufteilung einer rechenintensiven Aufgabe in viele kleine Jobs, sofern möglich, eine schnelle und effiziente Abarbeitung: Kleine, wenig anspruchsvolle Jobs passen oft zwischen zwei große, sich blockierende Jobs oder

können nebenläufig abgearbeitet werden. Für zentral organisierte Cluster wurde das Paket `BatchJobs` (Bischl u. a., 2012a) entwickelt, welches im anschließenden Abschnitt 5.3 vorgestellt wird.

Größere Rechenressourcen müssen typischerweise angekauft werden. Als Beispiel sei der Cloud-Computing Provider *Amazon Elastic Compute Cloud EC2* (Amazon.com, 2014) gegeben. Als dezentraler Zusammenschluss von Rechenknoten mit überschüssigen Ressourcen zeichnet Clouds deren Heterogenität und vergleichsweise hohen Kosten für Kommunikation und Speicherung aus. Die Größe des Netzwerks ist dabei flexibel, es kann dynamisch verkleinert oder vergrößert werden. Für große Daten, insbesondere zur Analyse von *big data*, bei der mehrere Peta-Byte an Daten prozessiert werden müssen, sind zudem Hadoop-Cluster (Apache Foundation, 2014) sehr beliebt. Für beide Plattformen existieren diverse Ansätze zur Anknüpfung von `R`, sie spielen jedoch für die Analyse von Genexpression nicht nur aufgrund der unklaren rechtlichen Situation zur Datensicherheit eine eher untergeordnete Rolle. Deshalb sei für eine weitere Übersicht auf Eddelbuettel (2014) verwiesen.

5.3 BatchJobs

Wie im vorherigen Abschnitt beschrieben, ist `BatchJobs` für durch Scheduler organisierte Cluster entwickelt worden.

Zunächst wird der Ablauf zur Nutzung solcher Cluster ohne das `BatchJobs`-Paket beschrieben. Dafür muss ein `R`-Skript erstellt werden, welches für die Stapelausführung durch `Rscript` bzw. `R CMD BATCH` geeignet ist. Unter anderem bedeutet das, dass sämtliche benötigte Eingaben geeignet übergeben werden. Hier existieren verschiedene Ansätze, etwa Kommandozeilenargumente oder das harte Kodieren von Einstellungen in Skript-Duplikaten. Beide Möglichkeiten sind händisch durchführbar für kleinere Experimente, erfordern jedoch eine Automatisierung mittels eines zusätzlichen Skriptes ab moderaten Experimentgrößen. Weiterhin muss das Endresultat der Berechnung auf Festplatte gespeichert werden, bevor der `R`-Prozess terminiert. Die Dateinamen stellen hier ein zusätzliches Problem dar, da die parallel laufenden Prozesse nicht schreibend auf dieselbe Datei zugreifen können. Entsprechend muss für jeden Prozess eine eindeutige Zeichenkette zur Ausgabe bestimmt werden. Nachdem alle Berechnungen abgeschlossen sind, was über Kommandozeilentools und reguläre Ausdrücke festgestellt werden kann, müssen alle Ausgabedateien wieder eingelesen und geeignet aggregiert werden. Aufgrund des Zwangs

zu eindeutigen Dateinamen in Kombination mit den Restriktionen des Dateisystems bezüglich der Anzahl und der Menge erlaubter Zeichen sind die Resultate oft schwierig den jeweiligen Parametrisierungen zuzuordnen. Fehlende Dateien, verursacht etwa durch fehlerhafte Programmierung, numerische Instabilitäten oder falsche Ressourcenspezifikation, bergen dabei besondere Herausforderungen für das Einsammeln der Ergebnisse.

Das beschriebene Vorgehen erfordert oft eine Umstrukturierung und Anpassung vorhandener Skripten zur Stapelausführung. Dies erschwert die Reproduzierbarkeit, welche in Abschnitt 5.5 ausführlich diskutiert wird, deutlich. Für Nutzer, welche keine oder wenig Erfahrung mit Linux und der Shell haben, stellt das gesamte Vorgehen eine oft unüberwindbare Hürde dar. Die Nutzung eines Clusters aus `R` heraus stand entsprechend im Mittelpunkt der Paketentwicklung.

5.3.1 Konfiguration

BatchJobs unterstützt derzeit fünf verschiedene Scheduling Systeme:

- PBS: Portable Batch System (Henderson, 1995), insbesondere die TORQUE Implementierung, in Dortmund in Betrieb auf LiDOng.
- SLURM: Simple linux utility for resource management (Yoo, Jette und Grondona, 2003), in Dortmund im Betrieb auf dem Cluster der Fakultät Statistik.
- LSF: Load Sharing Facility (Zhou, 1992), seit 2012 auch IBM Platform LSF genannt.
- SGE: Sun Grid Engine (Gentzsch, 2001), später Oracle Grid Engine, dann Univa Grid Engine.
- SSH: Eigene Implementierung. Basiert auf einem rudimentären Scheduler für Verbünde von mehreren über SSH erreichbare Maschinen mit einem gemeinsamen geteiltem Dateisystem.

Die ersten vier Systeme bieten jeweils eigene Kommandozeilentools zum Abschicken von Jobs. Die Kommandozeilenargumente unterscheiden sich dabei semantisch und inhaltlich, jedoch kann bei allen Systemen anstatt einer Vielzahl von Argumenten ein annotiertes Shell-Skript übergeben werden. Solch ein Shell-Skript umfasst das auszuführende Kommando und Informationen zur Ausführung. Diese Informationen sind mittels Kommentaren im jeweiligen Scheduler-Dialekt in die Datei kodiert. `BatchJobs` erstellt für den Benutzer eine solche Job-Datei anhand einer Vorlage. Die Vorlage muss einmalig

vom Benutzer erstellt werden, auf der `BatchJobs`-Website werden jedoch Basis-Vorlagen angeboten, welche für die meisten Systeme direkt lauffähig und bei Bedarf sehr einfach adaptierbar sind. Neben den Vorlagen existieren für jedes System `R`-Funktionen zum Absetzen, Terminieren und Auflisten von Jobs, entsprechend zugeschnitten auf das zugrundeliegende System und dessen Kommandozeilentools.

Die Wahl des Systems und der zu verwendenden Vorlage kann zwar in einer `R`-Sitzung interaktiv geschehen, jedoch ist dafür eine systemspezifische Konfigurationsdatei wesentlich praktischer. Solche Konfigurationsdateien können vom Systemadministrator global für alle Nutzer definiert werden, vom Nutzer für alle seine Experimente oder pro Projekt. Eine exemplarische Minimalkonfiguration ist durch

```
cluster.functions = makeClusterFunctionsSlurm("~/.slurm.tmpl")
default.resources = list(walltime = 60*60, memory = 1024)
```

gegeben. In der ersten Zeile werden das Scheduling-System (Slurm) und die dazu passende Vorlage ausgewählt. In der nächsten Zeile werden die Ressourcen für jeden Job, wenn nicht explizit anders angegeben, definiert. Zusätzlich können Optionen gesetzt werden, welche `BatchJobs` dazu veranlassen an definierten Ereignissen Status-Mails zu versenden. Weitere Optionen schalten Debug-Optionen oder teilweise experimentelle Workarounds ein. So kann `R` etwa dazu veranlasst werden, nach der Erstellung einer Datei so lange auf das Dateisystem zu warten, bis die gerade erstellte Datei auch tatsächlich gelistet wird. Andernfalls kann bei Netzwerkdateisystemen mit hoher Latenz ein reibungsloser Ablauf nicht garantiert werden.

Ein einfaches Template beinhaltet, wie bereits beschrieben, auskommentierte Anweisungen für den Scheduler sowie das auf dem Rechenknoten auszuführende Kommando (letzte Zeile):

```
#SBATCH --job-name=<%= job.name %>
#SBATCH --output=<%= log.file %>
#SBATCH --error=<%= log.file %>
#SBATCH --time=<%= resources$walltime %>
#SBATCH --ntasks=<%= resources$ntasks %>
#SBATCH --cpus-per-task=<%= resources$ncpus %>
#SBATCH --mem-per-cpu=<%= resources$memory %>

R CMD BATCH --no-save --no-restore "<%= rscript %>" /dev/stdout
```

Alles zwischen `<%=` und `%>` wird dabei durch `BatchJobs` mit den Werten der entsprechenden Variablen ersetzt.

5.3.2 Basis-Funktionalität

Auf organisierten Cluster-System erfolgt die Kommunikation zwischen dem Master (auf dem der Benutzer sich eingeloggt hat) und den Rechenknoten ausschließlich über das Dateisystem. Dies erfordert in `R` die Spezifikation eines Ordners, über welchen das Paket die Kommunikation der Prozesse abwickeln kann. Dazu wird in `BatchJobs` ein Registrierungs-Objekt erstellt, in dem solche Pfadinformationen kodiert sind.

```
reg = makeRegistry(id = "example", file.dir = tempfile())
```

Nachdem eine Registrierung erstellt ist, können mittels *Map* oder *Reduce* Jobs definiert werden. Ein Beispiel ist die Quadratur der ersten zehn natürlichen Zahlen:

```
ids = batchMap(reg, function(x) x^2, x = 1:10)
```

Die Funktion liefert einen Ganzzahl-Vektor mit Job-IDs zurück. Dadurch, dass jeder Job eine eindeutige Nummer zur Identifizierung besitzt, kann jede Operation auch auf Teilmengen von Jobs angewendet werden. Beispielsweise werden hier nur die ersten fünf Jobs an den Scheduler übermittelt:

```
submitJobs(reg, ids[1:5], resources = list(walltime=60))
```

Eine Übersicht über den aktuellen Stand der Berechnungen verschafft `showStatus()`:

```
showStatus(reg)


## Status for 10 jobs at 2015-03-02 15:54:29
## Submitted:  5 ( 50.00%)
## Started:    5 ( 50.00%)
## Running:    0 (  0.00%)
## Done:       5 ( 50.00%)
## Errors:     0 (  0.00%)
## Expired:    0 (  0.00%)
## Time: min=0.00s avg=0.40s max=1.00s
```

Zum Abfragen von Job-IDs basierend auf deren Status ist eine Familie von `find*`-Funktionen implementiert:

```
# Jobs mit erfolgreich abgeschlossenen Berechnungen
done = findDone(reg)

# Davon diejenigen Jobs mit Parameter x >= 4
findJobs(reg, ids = done, pars = (x >= 4))

## [1] 4 5
```

Letztlich steht die Aggregation der Ergebnisse aus, welche als Liste (`loadResults()`) oder durch ein allgemeines Reduce (`reduceResults()`) erfolgen kann:

```
unname(simplify2array(loadResults(reg)))

## [1]  1  4  9 16 25

reduceResults(reg, fun = function(aggr, job, res) c(aggr, res),
  init = numeric(0))

## [1]  1  4  9 16 25
```

Zusammenfassend besteht der Ablauf in `BatchJobs` also aus vier Phasen:

Init Erstellen der Registrierung,

Definition Definieren von Jobs,

Submit Abschicken der Jobs an den Scheduler, und

Reduce Einsammeln der Ergebnisse.

Die ersten drei Schritte sind in der Funktion `batchMapQuick()` zusammengefasst. Ein Kombination aus allen vier Schritten ist in Zusatzpaketen verwirklicht, welche in Abschnitt 5.3.4 vorgestellt werden.

5.3.3 Debugging

Einer der größten Nachteile beim verteilten parallelen Rechnen stellen die, im Vergleich mit der lokalen und sequentiellen Ausführung, verminderten Möglichkeiten zur Fehleridentifikation dar. So ist etwa die zum Debuggen essentielle Funktion `traceback()` beim verteiltem Rechnen nicht verfügbar. Um diesen Nachteil abzuschwächen legt `BatchJobs` Protokolle sämtlicher Ausgaben der `R`-Prozesse an. Die Funktion `findErrors()` kann zur Bestimmung fehlerhafter Job-IDs genutzt werden. Diese IDs können dann an die Funktion `showLog()` zur Anzeige der entsprechenden Log-Datei übergeben werden. Mittels `grepLogs()` lassen sich alle Protokolle nach einer Zeichenkette, etwa einer Warnmeldung, durchsuchen. Die Funktion `testJob()` erlaubt darüber hinaus die lokale Ausführung eines einzelnen Jobs in einem neuen `R`-Prozess. Wird in `testJob()` das Argument `external=FALSE` gesetzt, wird der Job in der laufenden `R`-Sitzung ausgeführt, wodurch das übliche Debugging mittels `traceback()` oder `browser()` verfügbar wird. Weiterhin sind alle Experimente in `BatchJobs` deterministisch. Jedem Job ist ein individueller Startwert für den Zufallszahlengenerator zugewiesen (siehe Abschnitt 5.5). Dadurch können Fehler stets reproduziert und analysiert werden.

5.3.4 Standard Map-Funktionalität

Durch den mehrschrittigen Ablauf eines Experiments (Init, Define, Submit und Reduce, siehe vorherigen Abschnitt 5.3.2) ist das Paket sehr flexibel und kann daher auch in ungewöhnlichen Szenarien eingesetzt werden. Jedoch ist eine reine und „gewohnte“ Map-Operation als Zusammenlegung der Einzelschritte oft praktischer. Dabei ist das Verhalten bei Fehlern das Hauptproblem, da zunächst ein Rückgabewert für fehlerhafte Funktionsaufrufe definiert werden muss. Zudem muss ein temporäres Verzeichnis erstellt werden und sichergestellt werden, dass es (a) wieder entfernt wird, wenn es nicht mehr benötigt wird, und (b) dieses Verzeichnis auf allen Knoten des Netzwerkes verfügbar ist.

Solch eine Map-Funktionalität ist im Paket `BiocParallel` (Morgan, Thompson und Lang, 2014) auf Bioconductor (Gentleman u. a., 2004) zu finden. Die Kernfunktionen von `BiocParallel` umfassen `bplapply()` und `bpmapply()` als parallele Varianten von `lapply()` bzw. `mapply()`. Die Art der Parallelisierung muss mittels eines speziellen `Param`-Objekts vor dem entsprechenden Aufruf spezifiziert werden. Die Unterstützung von `BatchJobs` wurde im Rahmen des *Google Summer of Code 2013* (GSOC2013) in die

bestehende Struktur, welche bereits die Pakete `multicore` und `snow` (beide jetzt fusioniert zu `parallel`) sowie `foreach` (Revolution Analytics und Weston, 2014) unterstützte, eingepflegt. Außerdem ist während des GSOC2013 ein Mechanismus zur Wideraufnahme von partiellen Berechnungen implementiert worden. Durch die enge Verzahnung von Bioconductor-Paketen findet `BiocParallel` bereits in zahlreichen bioinformatischen Paketen Verwendung.

Ein ähnlicher Ansatz ist in `parallelMap` (Bischl und Lang, 2014) verfolgt worden. Der grundsätzliche Funktionsumfang ist dabei sehr ähnlich. Genau wie bei `BiocParallel` oder auch `foreach` muss zunächst die Art der Parallelisierung registriert werden, um anschließend Map-artige Operationen auszuführen. Die Abgrenzung gegenüber dem Paket `BiocParallel` liegt primär in der geringeren Integrationsmöglichkeit in CRAN Paketen. Das `foreach`-Paket grenzt sich dadurch von den anderen beiden Paketen ab, dass die Operation in einer Schleifen-Struktur geschrieben wird. Der Ansatz, eine Schleife zu imitieren führte jedoch zumindest in der Vergangenheit oft zu Problemen beim Nachschlagen von Variablen, da hier mittels einer Heuristik die benötigten Variablen erkannt und automatisch an die Rechenknoten exportiert werden sollen. Dies klappt jedoch nicht immer fehlerfrei, unter gewissen Umstände werden benötigte Variablen nicht exportiert oder zur Ausführung irrelevante Objekte exportiert. Der erste Fall kann nicht mit allen Systemen reproduziert werden, weshalb diese Art von Fehlern auch in veröffentlichten Paketen anzutreffen ist. Der zweite Fall fällt nur bei der Verwendung von großen Objekten wie Genexpressionsmatrizen ins Gewicht, kann dann aber auch das ganze System zum Erliegen bringen. Um beide Fälle zu korrigieren gibt es die Möglichkeit, Objekte explizit zu exportieren oder vom Export auszuschließen, was sich in der Praxis aber oft als unpraktikabel erweist.

Alle genannten Pakete sind insbesondere für Paketentwickler sehr interessant. In `R` gibt es derzeit keine standardisierte Möglichkeit, die Art der parallelen Ausführung innerhalb eines Paketes zu adaptieren. Oft findet sich in Paketen daher lediglich für ein Backend Unterstützung, wobei zusätzliche Parameter zur parallelen Ausführung durchgeschleift werden müssen. Mittels des beschriebenen Registrierungs-Mechanismus kann der Nutzer dagegen fein-granular Einfluss auf Art und Parametrisierung der Parallelisierung nehmen. Auch den Paketentwicklern wird viel Arbeit erspart, da hier auf ein gut getestetes System zurückgegriffen werden kann. Hervorzuheben ist auch, dass `parallelMap` die Vergabe von eindeutigen Namen für Map-Operationen erlaubt. Bei geschachtelten Schleifen kann der Nutzer so etwa in Abhängigkeit von den Eingabedaten und der verfügbaren Hardware wählen, ob beispielsweise bei geschachtelten Map-Operationen die äußere oder innere

Schleife parallelisiert wird.

5.4 BatchExperiments

Computerexperimente sind häufig integraler Bestandteil statistischer Arbeit. Für die meisten angewandten statistischen Neuentwicklungen müssen umfangreiche Vergleichsstudien zwischen Modellen angefertigt werden. Aber auch für eine einfache Datenanalyse müssen oft verschiedene Vorverarbeitungen, Modelle und Parametrisierungen miteinander verglichen werden. Dies kann prinzipiell über ein einfaches Map erfolgen, indem über Listen mit Trainingsmengen, Modellen und Parametrisierungen iteriert wird. Allerdings hat dies in der Praxis viele Nachteile. So wird die Funktion und deren Eingabe sehr schnell sehr unübersichtlich und damit fehleranfällig. Zudem ist das Kombinieren der Einzelergebnisse oft sehr aufwändig. Außerdem bewirken kleine Änderungen am Versuchsaufbau, etwa die Änderung des Wertebereichs eines Parameters oder die Aufnahme eines weiteren Modells, dass alle Ergebnisse entweder neu berechnet oder aufwändig manuell miteinander kombiniert werden müssen.

Zur Lösung einer großen Klasse von Computerexperimenten wurde daher in dem Paket `BatchExperiments` eine Abstraktion geschaffen, welche auf `BatchJobs` aufbaut und somit direkt von der Parallelisierung profitiert. In `BatchExperiments` können allgemeine Probleme und Algorithmen definiert und parametrisiert miteinander kombiniert werden. Dies ist in Abbildung 5.1 schematisch dargestellt. Probleme bestehen aus Gründen der Effizienz aus zwei Teilen, einem statischen, unveränderlichen Part (z. B. einem `data.frame`) und einer dynamischen Problemfunktion. Letztere bekommt den statischen Part, sofern definiert, als Eingabe. Ein übliches Beispiel wäre hier eine Resampling-Funktion, welche aus dem statischen Teil Trainingsdatensätze generiert. Die dynamische Problemfunktion ist jedoch sehr frei definiert, sie kann auch Zufallszahlen generieren, eine Datenbankabfrage durchführen oder eine Datei lesen. Die Rückgabe dieser Funktion dient als Probleminstanz und wird an den Algorithmus, zusammen mit dem statischen Part, weitergereicht. Der Algorithmus ist dabei wieder eine beliebige `R`-Funktion, wobei die Rückgabe als Resultat des Jobs verstanden wird. Sowohl die dynamische Problemfunktion als auch die Algorithmusfunktion kann mit einem Design verknüpft und somit parametrisiert werden. Die Designs werden an die Funktion `addExperiments()` gegeben, welche daraus Jobs für die `BatchJobs`-Infrastruktur generiert. Die Art, wie beide Pakete zu verwenden sind, unterscheidet sich entsprechend nur in der Art, wie Jobs definiert

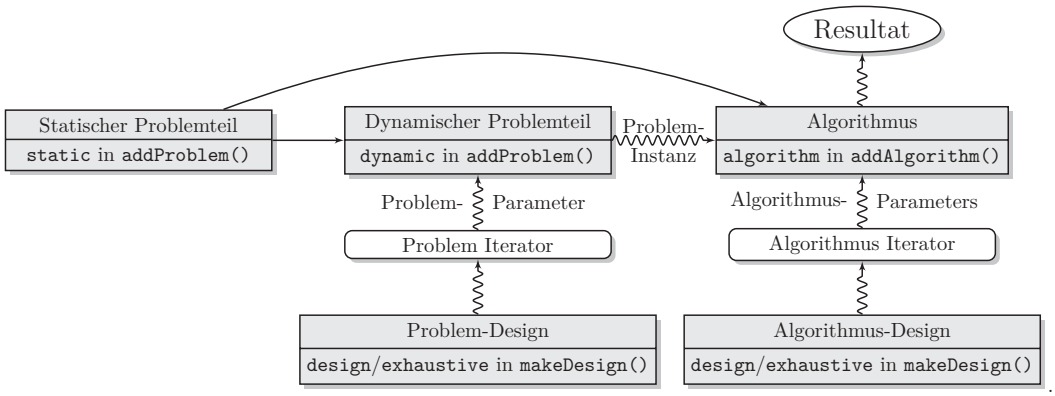

Abbildung 5.1: Zusammenhang der `BatchExperiments`-Funktionen zur Definition von Jobs. Exportierte Funktionen sind grau hinterlegt, intermediäre interne Objekte in weiß. Gerade Linien deuten die direkte Übergabe des Objekts bzw. der Funktion an, eine geschlängelte Line deren evaluiertes Resultat. Grafik adaptiert aus Bischl u. a. (2012a).

werden. Die Organisation der Berechnung, also das Abschicken oder Beenden der Jobs, die Arbeit mit Untermengen oder das Ressourcenmanagement, sind identisch. Dies wird ebenfalls durch Abbildung 5.2 veranschaulicht, welche die wichtigsten Funktionen beider Pakete zusammenfasst. Dabei wird insbesondere deutlich, dass sich beide Pakete primär in der Art unterscheiden, wie Jobs definiert werden. Natürlich besitzt die Registry eine leicht andere Struktur, weshalb ein anderer Konstruktor benötigt wird. Ansonsten kann `BatchExperiments` auf alle Funktionen von `BatchJobs` zurückgreifen und bietet darüber hinaus noch einige Spezialisierungen an.

Das folgende Beispiel illustriert die Verwendung von `BatchExperiments` und demonstriert gleichzeitig die Verwendung der Funktion `reduceResultsExperiments()`, einer Spezialisierung, welche das systematische Aggregieren besonders einfach macht. Dazu wird zunächst eine Registry angelegt.

```
library(BatchExperiments)
library(mlr)
reg = makeExperimentRegistry(id = "example", file.dir = tempfile())
```

Anschließend wird ein Überlebenszeitproblem als Holdout-Stichprobe des Datensatzes „Wisconsin Prognostic Breast Cancer Data“, zu finden im Paket `TH.data` (Hothorn, 2014), definiert. Die Daten werden einfachheitshalber direkt zu einem `mlr`-Task konvertiert. Zuvor wird dazu die Spalte des Zensierungsindikators von einem Faktor zu einer logischen

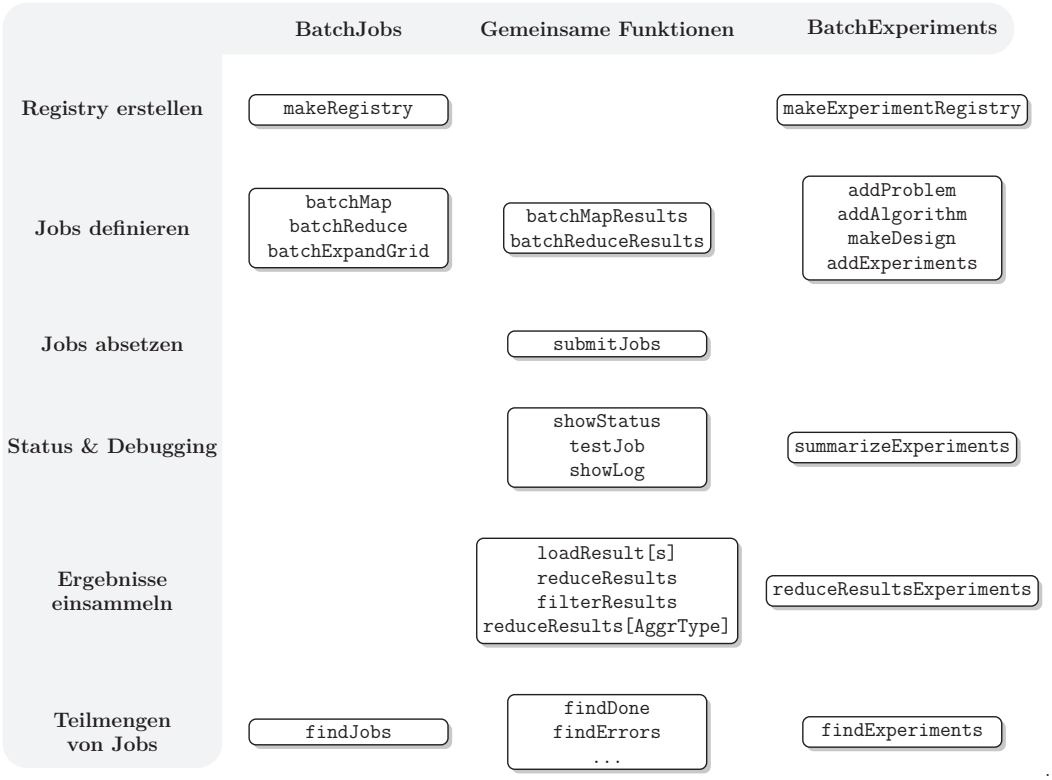

Abbildung 5.2: Gemeinsamkeiten und Unterschiede zwischen den Paketen `BatchJobs` und `BatchExperiments` in dem Repertoire angebotener Funktionen. Die hier gelisteten Funktionen stellen die Basismenge der wichtigsten Funktionen dar, nicht alle implementierten. Grafik adaptiert aus Bischl u. a. (2012a).

Variable konvertiert und alle Beobachtungen entfernt, welche fehlende Werte in mindestens einer Kovariablen haben. Für das Resampling wird ebenfalls auf `mlr` zurückgegriffen. Dies erlaubt eine einfache Stratifikation auf dem Zensierungsindikator und stellt somit sicher, dass in jeder Trainingsaufteilung mindestens ein Ereignis ist.

```
data("wpbc", package = "TH.data")
wpbc$status = (wpbc$status == "R")
wpbc = wpbc[complete.cases(wpbc), ]
task = makeSurvTask(id = "wpbc", data = wpbc, target = c("time", "status"))
addProblem(reg,
  id = "wpbc",
  static = task,
  dynamic = function(static) {
    desc = makeResampleDesc("Holdout", split = 0.67, stratify = TRUE)
```

```
    makeResampleInstance(desc, task = static, size = 1)
  },
  seed = 1
)
```

Anschließend wird als Algorithmus ein Überlebenszeitwald gewählt. Das Design definiert ein Gitter-Layout auf den Hyperparametern `nodesize` und `mtry`.

```
addAlgorithm(reg,
  id = "rfsrc",
  fun = function(job, static, dynamic, ...) {
    lrn = makeLearner("surv.randomForestSRC", ...)
    resample(lrn, task = task, resampling = dynamic)$aggr
  }
)
pars = list(mtry = c(1, 3, 5, 7), nodesize = c(3, 5, 7))
rfsrc.design = makeDesign("rfsrc", exhaustive = pars)
```

Das Algorithmen-Design muss zur Erstellung von Compute-Jobs lediglich noch mit einem Problem verknüpft werden. Da das Problem frei von Parametern ist, kann hier direkt die Problem-ID angegeben werden.

```
addExperiments(reg,
  prob.designs = "wpbc",
  algo.designs = rfsrc.design,
  repls = 10)
```

Es wurden zehn Replikationen gewählt. Durch das vorherige Setzen des Problem-Seeds erhalten etwaige zusätzliche Algorithmen identische Probleminstanzen per Replikation, siehe dazu auch folgenden Abschnitt 5.5 zur Reproduzierbarkeit. Eine Übersicht über die definierten Jobs erlaubt die Funktion `summarizeExperiments()`:

```
summarizeExperiments(reg, show = c("prob", "algo", "mtry", "nodesize"))
```

```
##    prob  algo mtry nodesize .count
## 1  wpbc rfsrc    1        3     10
## 2  wpbc rfsrc    1        5     10
## 3  wpbc rfsrc    1        7     10
## 4  wpbc rfsrc    3        3     10
```

```
## 5  wpbc rfsrc    3        5     10
## 6  wpbc rfsrc    3        7     10
## 7  wpbc rfsrc    5        3     10
## 8  wpbc rfsrc    5        5     10
## 9  wpbc rfsrc    5        7     10
## 10 wpbc rfsrc    7        3     10
## 11 wpbc rfsrc    7        5     10
## 12 wpbc rfsrc    7        7     10
```

Nach der Berechnung, angestoßen durch `submitJobs()`, können alle Ergebnisse bequem in einem `data.frame` zusammengefasst werden:

```
res = reduceResultsExperiments(reg)
res[c(1:3, 118:120), ]
```

```
##      id prob  algo mtry nodesize repl cindex.test.mean
## 1     1 wpbc rfsrc    1        3    1        0.5911528
## 2     2 wpbc rfsrc    1        3    2        0.6084584
## 3     3 wpbc rfsrc    1        3    3        0.6508172
## 118 118 wpbc rfsrc    7        7    8        0.5662100
## 119 119 wpbc rfsrc    7        7    9        0.5657439
## 120 120 wpbc rfsrc    7        7   10        0.5990640
```

Eine Aggregation über die Replikationen ist durch das Paket `data.table` (Dowle, Lianoglou und Srinivasan, 2014) einfach zu erstellen.

```
library(data.table)
dt = as.data.table(res)
dcast.data.table(dt, mtry ~ nodesize, fun.aggregate = mean,
  value.var = "cindex.test.mean")
```

```
##    mtry         3         5         7
## 1:    1 0.6047774 0.6047822 0.5990697
## 2:    3 0.5892747 0.5939484 0.5931244
## 3:    5 0.5780458 0.5886395 0.5906876
## 4:    7 0.5708976 0.5850207 0.5914390
```

Ein Wert von `mtry == 1` und `nodesize == 5` scheint die beste Einstellung zu sein, wobei die Änderung von `mtry` einen größeren Einfluss auf die Vorhersagegüte des Überlebens-

zeitwaldes zu haben scheint als Variationen des Parameters `nodesize`. Dies ist natürlich keine repräsentative Studie, aber sie demonstriert, wie einfach systematische Parameterstudien oder Benchmarkstudien mit Hilfe von `BatchExperiments` programmierbar sind. Desweiteren zeigen sich vielversprechende Synergien zwischen `BatchExperiments` und `mlr`, welche beide durch eine engere Verzahnung profitieren können. Erste Vorarbeiten dazu sind bereits im Repository von `mlr` auf GitHub zu finden.

5.5 Reproduzierbarkeit

Zur Sicherstellung einer späteren Reproduzierbarkeit von Computerexperimenten in `R` sind folgende vier Punkte unverzichtbar:

1. Daten und Quellcode müssen bereitgestellt werden bzw. quelloffen verfügbar sein. Dazu bietet sich ein `R`-Paket als kombiniertes Datenformat für Projekte an. Da jedoch nicht jedes Paket die Anforderungen von CRAN hinsichtlich Qualität oder allgemeiner Nützlichkeit erfüllt, bietet sich insbesondere während der Erstellung von Paketen die Plattform GitHub als aktueller de-facto Standard zum Bereitstellen von freier Software an. Selbstverständlich müssen alle Abhängigkeiten ebenfalls quelloffen verfügbar sein und ihre Versionsnummern (z. B. mittels `sessionInfo()`) zusammen mit Resultaten dokumentiert werden.

2. Die erstellte Software muss sich unter Versionsverwaltung befinden. Nur durch eine Versionverwaltung wie Subversion oder Git kann nachvollzogen werden, welche Resultate durch eventuelle Fehler im Programmcode beeinflusst worden sind.

3. Die Struktur des Quellcodes muss verständlich organisiert sein und ggf. durch Kommentare und zusätzliche Dokumentation ergänzt werden. Neben Quellcode und Daten kann ein `R`-Paket auch Dokumentation und zusätzliches Material, zum Beispiel in Form von Vignetten, enthalten. Zur Dokumentation von quelloffener Software hat sich für fast alle Programmiersprachen ein System durchgesetzt, welches aus Annotationen im Quellcode automatisch Dokumentation erzeugt. Als Vorbild gilt zumeist Doxygen (Van Heesch, 2004), woran sich auch das `R`-Pendant `roxygen2` (Wickham, Danenberg und Eugster, 2014) orientiert.

4. Bei stochastischen Experimenten muss der Zufallszahlengenerator (RNG) durch einen Startwert (Seed) initialisiert werden. Konkret muss es sich um einen Pseudo-RNG handeln, also einem Generator, welcher nur abhängig von seinem Zustand

einen reproduzierbaren Strom von Zahlen generiert. Zur Sicherstellung der Reproduzierbarkeit muss der Zustand nun nicht wie sonst üblich durch Zufall aus der Peripherie (z. B. Systemzeit oder Netzwerkaktivität) initialisiert werden, sondern explizit und deterministisch beim Start des Experiments gesetzt werden. In `R` erfolgt die Initialisierung über `set.seed(x)` mit $x \in \mathbb{N}_+$. Zudem muss streng genommen ein Pseudo-RNG mittels `RNGkind()` explizit ausgewählt werden. `R` benutzt seit einigen Jahren den Mersenne-Twister von Matsumoto und Nishimura (1998) als Voreinstellung.

Beim parallelen Rechnen sind die Punkte (3) und (4) erheblich schwieriger als beim sequentiellen Rechnen. Die Organisation des Quellcodes muss oft an das ausführende System adaptiert werden, um eine effiziente Ausführung zu gewährleisten. Dazu wird die logische Struktur zugunsten des Ressourcenverbrauchs aufgebrochen, wodurch der Code schwieriger verständlich wird. Oft müssen auch Anpassungen an das System vorgenommen werden, was für eine Reproduktion auf anderen Systemen erheblichen Aufwand bedeutet. Die in dem vorherigen Abschnitt vorgestellten Pakete zur parallelen Ausführung einfacher Map-Operationen reduzieren die Häufigkeit, in der solche Eingriffe nötig sind. Für große, umfangreiche Computerexperimente bietet `BatchExperiments` aus Abschnitt 5.4 ein umfangreiches Framework, welches eine Adaption an das Ausführungssystem in den meisten Fällen unnötig macht.

Das zweite große Problem stellen die Zufallszahlen dar. Zwar stellen die `R`-Pakete zur Parallelisierung spezielle Funktionen zum setzen von Seeds bereit, jedoch hängen die Zufallszahlenströme sowohl von der Länge der Eingabe als auch von der Anzahl der eingesetzten Recheneinheiten ab. Beispielsweise liefert eine stochastische Funktion auf einer vierelementigen Eingabe bei der Berechnung mittels `parallel::mclapply()` auf zwei Kernen andere Ergebnisse als bei der Berechnung auf vier Kernen.

Die Pakete `BatchJobs` und `BatchExperiments` umgehen die RNG-Problematik durch die Registry. Bei Definition eines Jobs wird immer ein fester und eindeutiger Seed zugeteilt und persistent in der Datenbank gespeichert. In `BatchJobs` ist dieser über das Inkrement eines globalen Registry-Startwerts definiert. Der pro Job eindeutige Seed wird direkt vor Funktionsaufruf gesetzt und der RNG, nachdem die Funktion terminiert ist, in seinen vorherigen Zustand zurückversetzt. In `BatchExperiments` wird ein ähnlicher Mechanismus genutzt, es existiert jedoch zusätzlich eine optionale Problem-Synchronisation: Die Instanz-Generierung kann anstatt mit einem Job-Seed auch mit einem Problem-Seed laufen, welche mit der Replikationsnummer inkrementiert wird. Somit

können Algorithmen einfach miteinander verglichen werden, da sie in jeder Replikation auf identischen Instanzen arbeiten.

6 Automatische Modellselektion in der Überlebenszeitanalyse

In diesem Kapitel wird im ersten Abschnitt 6.1 zunächst die Datenakquise und Datenvorverarbeitung thematisiert. Eine kurze deskriptive Beschreibung gibt dabei einen Überblick über die verwendeten Datensätze. Der geplante Versuchsaufbau schließt im Abschnitt 6.2 an. Dies beinhaltet unter anderem eine ausführliche Beschreibung von vier Optimierungsstrategien, welche sowohl untereinander als auch mit üblichen Referenzmodellen verglichen werden sollen. Die Ergebnisse der Vergleiche werden im folgenden Abschnitt 6.3 zunächst grafisch, anschließend tabellarisch über Rangstatistiken vorgestellt.

6.1 Daten

Die für diese Arbeit zugrundeliegenden Daten wurden durch eine Hochdurchsatz-Technologie der Systembiologie, die sogenannten DNA-Microarrays oder Gen-Chips (Schena u. a., 1995), erhoben. Diese bestehen aus Tausenden komplementären DNA-Strängen, welche auf wenigen Quadratzentimetern untergebracht sind. Eine zu messende Zellprobe wird so präpariert, dass ihre cDNA mit fluoreszierenden oder radioaktiven Stoffen verbunden wird, um sie auf diese Weise zur späteren Wiedererkennung zu markieren. Die cDNA bindet an die entsprechenden komplementären Stränge auf dem Chip. Überschüssiges Zellmaterial wird abgewaschen, dabei verbleiben nur gebundene Stränge auf dem Chip. Dies ist in Abbildung 6.1 veranschaulicht. Die induzierte Helligkeit erlaubt Rückschlüsse auf die Menge bzw. Bindungsstärke und somit auf die Genaktivität. Pro komplementären DNA-Strang auf dem Chip kann also ein numerischer Wert für die Helligkeit abgeleitet werden. Diese Helligkeitswerte können nach einer Normalisierung direkt als metrische Variablen in der statistischen Modellierung berücksichtigt werden. Microarrays erlauben somit eine ökonomische Messung von vielen Tausend Genen gleichzeitig.

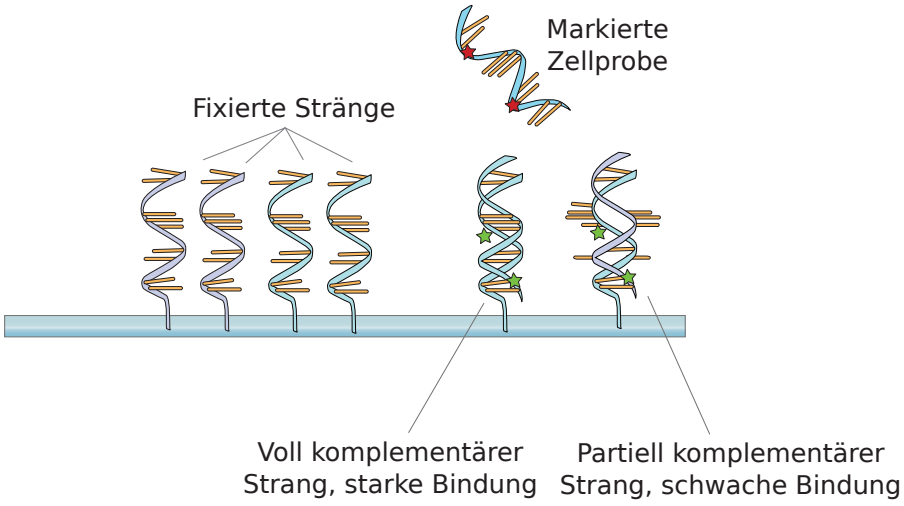

Abbildung 6.1: Funktionsweise eines DNA-Microarrays. Abbildung adaptiert von `http://en.wikipedia.org/wiki/File:NA_hybrid.svg`

Durch Datenbanken wie *Gene Expression Omnibus* (GEO) (Edgar, Domrachev und Lash, 2002) können Genexpressionsdaten und zugehörige klinische Kovariablen zur freien Verfügung publiziert werden. In den folgenden Unterabschnitten werden Kriterien zur Auswahl, zur notwendigen Vorverarbeitung und zur Aufbereitung beschrieben.

6.1.1 Kriterien zur Datensatzwahl

Als Grundlage für die Auswertungen dient eine Zusammenstellung von insgesamt sieben frei verfügbaren Lungenkrebsdatensätzen. Sechs von ihnen wurden aus der GEO-Datenbank extrahiert, ein weiterer ist über das Supplement der Publikation Shedden u. a. (2008) verfügbar. Hauptaugenmerk bei der Auswahl lag auf einer möglichst hohen Homogenität. Gleichzeitig sollte die Anzahl der Beobachtungen groß genug sein, um sinnvoll modellieren zu können. Konkret wurde die Auswahl durch folgende Charakteristika beschränkt:

Überlebenszeiten Es wurden ausschließlich Datensätze mit Überlebenszeittyp *Overall Survival* (OS) selektiert und ausgewertet, da diese in den meisten Datensätzen vorhanden war. Der Endpunkt stellt in diesem Fall einfach den Tod des Patienten dar. Dabei gilt zu beachten, dass auch beispielsweise Unfälle als krankheitsrelevantes Ereignis zählen. Dies kann unter Umständen die Daten etwas verfälschen, andererseits werden Nebenwirkungen wie beispielsweise Schwindel (welcher dann zu

einem Unfall führt) implizit erfasst. Zudem handelt es sich bei OS um eine relativ unstrittige und klare Definition – eine allgemein akzeptierte Standardisierung der Definition von Endpunkten existiert nicht (siehe dazu etwa Hudis u. a. (2007)).

Klinische Kovariablen Klinische Kovariablen enthalten in der Regel eine Vielzahl von fehlenden Werten. Um die Anzahl der Datensätze als auch die dort enthaltenen jeweiligen Patientenzahlen zu maximieren, wurden daher nur wenige klinische Kovariablen selektiert, welche sich aber in vielen vorherigen Studien als krankheitsrelevant herausgestellt haben. Es handelt sich hier um die Variablen „sex“ (Geschlecht), „histology“ (durch Biopsie ermittelter Tumortyp) sowie „age“ (Alter). Eine weitere wichtige Variable, welche anzeigt ob der Patient bzw. die Patientin raucht oder bis vor kurzem geraucht hat, konnte aufgrund fehlender Standardisierung sowie vieler fehlender Werte nicht aufgenommen werden. Die Ausprägungen der klinischen Kovariablen sind nach Datensatz aufgeschlüsselt in Anhang A in den Tabellen A.9 bis A.11 zu finden. Dabei ist zu beachten, dass in einigen Fällen die Kohorte konstant in ihrem histologischen Befund war, also etwa nur Patientinnen und Patienten mit einem bestimmten Tumortyp in die Studie aufgenommen wurden (vergleiche Tabelle A.10 im Anhang A). In diesen Fällen wurde diese Variable vor den Modellanpassungen entfernt. Dies stellt für viele Lernverfahren eine technische Notwendigkeit dar, andernfalls würden Abbrüche aufgrund numerischer Instabilitäten auftreten.

Genetische Kovariablen Die genetischen Kovariablen wurden auf 2 Typen von Microarrays gemessen: „HG-U133-A“ und „HG-U133-Plus2“. Die Analysen sind auf die 22 277 *Probe Sets* beschränkt, welche auf beiden Chiptypen erfasst werden. Obwohl nicht immer jedem Probe Set ein Gen exakt zugeordnet werden kann, werden die Begriffe Probe Set und Gen im Folgenden der Einfachheit halber synonym verwendet.

Einen Überblick über die Verteilung der Überlebenszeiten gibt Abbildung 6.2, wobei Tabelle 6.1 die Grafik um die Anzahl der Beobachtungen und Ereignisse der einzelnen Kohorten ergänzt. Bei Betrachtung der geschätzten Überlebenszeitfunktionen fallen zunächst zwei Aspekte auf:

- Patientinnen und Patienten der Kohorte GSE31210 besitzen eine vergleichsweise gute Prognose. Dies wird durch die geringe Anzahl an Ereignissen (vergleiche Tabelle 6.1) gestützt.

- Die maximale Nachbeobachtungszeit unterscheidet sich deutlich, am stärksten im Vergleich zwischen Kohorte GSE14814 (9.3 Jahre) und der Kohorte GSE30219 (19.8 Jahre).

Diese Unterschiede können etwa durch verschiedene Kriterien zur Studienaufnahme und damit Unterschiede im anfänglichen Krankheitsstatus bzw. im typischen Krankheitsverlauf begründet sein. So zeigt etwa Tabelle A.10 auf, dass einige Studien verschiedene Tumortypen beinhalten, andere scheinen sich auf einen bestimmten Tumortyp zuvor festgelegt zu haben. Auch bei der Verteilung des Geschlechts, siehe Tabelle A.11, sind deutliche Unterschiede in der Zusammensetzung zu finden.

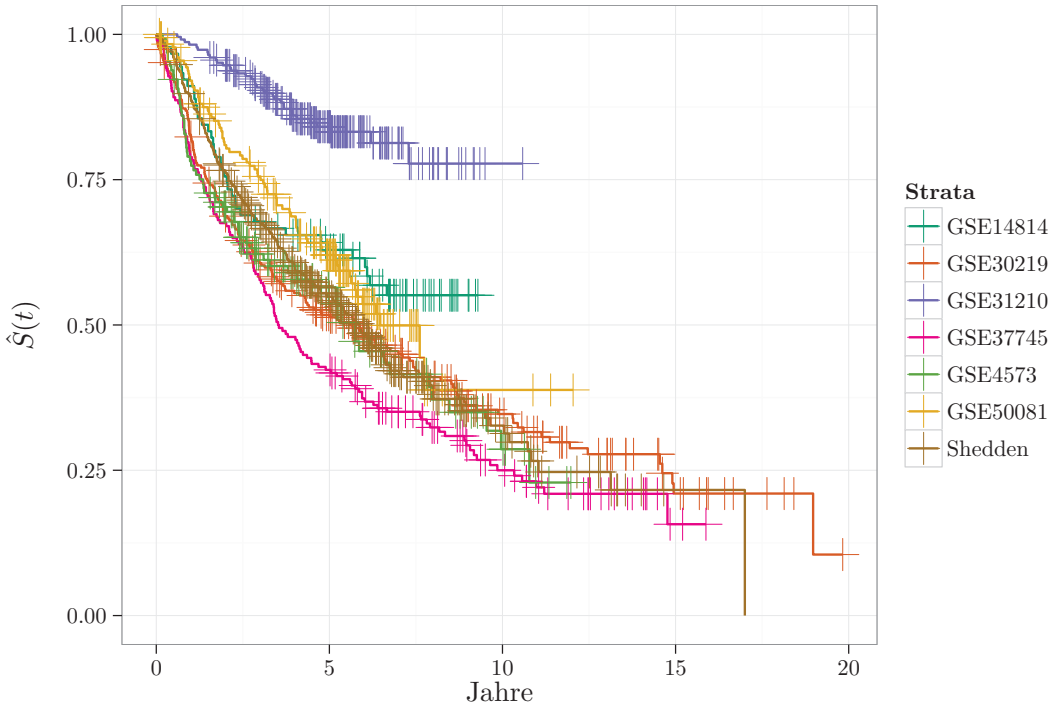

Abbildung 6.2: Kaplan-Meier-Schätzer der verwendeten Lungenkrebsdatensätze. Vertikale Linien kennzeichnen Zensierungen.

6.1.2 Vorverarbeitung

Bei dem Verfahren *Robust Multichip Average* (RMA) (Irizarry u. a., 2003) werden alle Erhebungen aneinander ausgeglichen und standardisiert. Durch dieses Vorgehen wird eine Abhängigkeit zwischen den Patienten eingefügt, was zur Folge hat, dass

Datensatz	Chip	Beobachtungen	Ereignisse
GSE14814	HG-U133-A	90	38
GSE4573	HG-U133-A	129	67
GSE50081	HG-U133-Plus2	181	75
GSE37745	HG-U133-Plus2	194	143
GSE31210	HG-U133-Plus2	226	35
GSE30219	HG-U133-Plus2	268	170
Shedden	HG-U133-A	442	236
Insgesamt	–	1530	764

Tabelle 6.1: Übersicht über den Chiptyp, die Anzahl Patienten und die Anzahl Todesfälle der verwendeten Lungenkrebsdatensätze.

(a) die RMA Normalisierung bei Veränderung der Stichprobe, etwa der Hinzunahme neuer Patienten, wiederholt werden muss, und

(b) in einem statistischen Experiment die Vorverarbeitung in jedem Schritt des äußeren Resamplings durchgeführt werden müsste.

Die genetischen Eingabevariablen der beiden Chiptypen wurden daher durch *frozen RMA* (McCall, Bolstad und Irizarry, 2010) normalisiert. Das heißt, alle Beobachtungen wurden einzeln anhand einer Referenzstichprobe standardisiert. Für den Datensatz GSE50081 wurden zudem Faktorstufen in der Variable Histologie zusammengelegt: Die insgesamt sieben Ausprägungen mit Stufe „largecell“ wurden mit den vier Ausprägungen mit Stufe „other“ fusioniert. Faktorvariablen mit so wenig Beobachtungen bieten in der Regel keine nutzbaren Informationen. Außerdem erschwert sich die Modellierung erheblich, da das Vorhersagen unbekannter Faktorstufen in den meisten Implementierungen mit einem Abbruch der Ausführung quittiert wird.

6.2 Aufbau des statistischen Experiments

In diesem Abschnitt wird zunächst der experimentelle Aufbau zur Validierung der Auswertungsstrategien erläutert. Eine Beschreibung des Hyperparameterraums zur Optimierung schließt sich an. Daran anschließend werden Einstellungen des verwendeten Surrogat-Modells sowie der Modell-basierten Optimierung thematisiert.

6.2.1 Validierung

Zur unverzerrten Beurteilung der Güte einer Modell-Konfiguration muss die Validierung unbedingt auf einem unabhängigem Testdatensatz erfolgen. Die Wahl des Trainings- und Testdatensatzes hat selbstverständlich einen Einfluss auf die resultierende Konfiguration und Güte, daher muss für einen Vergleich von Optimierungsverfahren stets ein äußeres Resampling verwendet werden. Oft wird dazu eine zehnfache Kreuzvalidierung benutzt. Für die Validierung von Überlebenszeitmodellen werden von den Gütemaßen jedoch recht viele Beobachtungen benötigt, und der verwendete Konkordanzindex stellt hier keine Ausnahme dar. Da lediglich die Reihenfolge der Überlebenszeiten im Testdatensatz evaluiert wird, müssen hier insbesondere auch mehrere unzensierte Ereigniszeiten vorliegen. Entsprechend wurde von der zehnfachen Kreuzvalidierung abgewichen und stattdessen eine fünffache Kreuzvalidierung durchgeführt. Diese ist zusätzlich an den Ereignissen stratifiziert, so dass einerseits in jedem Trainings- und Testdatensatz in etwa identische Zensierungsraten vorliegen, und andererseits die Validierung der Güte überhaupt technisch gewährleistet ist. Die Tatsache, dass die fünffache Kreuzvalidierung nicht mehrmals wiederholt worden ist, um somit Voraussetzungen für statistische Signifikanztests zu schaffen, ist den hohen Ressourcenanforderungen geschuldet.

Die mittels Modell-basierter Optimierung gefundenen Konfigurationen müssen anhand alternativer Herangehensweisen bewertet werden. Für den gegebenen Parameterraum kommt das *Iterated F-Racing* (López-Ibánez u. a., 2011) in Frage, welches bereits in Lang u. a. (2015) auf Überlebenszeitdaten angewendet worden ist. Allerdings ist das Verfahren in der aktuellen Implementierung schlecht parallelisierbar, so dass eine Anwendung bei gegebenem Budget, der Anzahl an Datensätzen und der Größe des Hyperparameterraums nicht möglich ist. Ein genereller Vergleich zwischen F-Racing und MBO steht derzeit noch aus, jedoch erfordert dieser auch systematische Variation des Hyperparameterraums sowie weiterer Parameter, welche die Optimierung direkt betreffen, um eine sinnvolle Aussage treffen zu können. Dies sollte jedoch letztlich besser auf kleineren Datensätzen mit geringeren Ressourcenanforderungen analysiert werden. Stattdessen erfolgt in dieser Arbeit ein Vergleich mit drei alternativen Analysestrategien.

Random Search Die *Random Search* (RS) (Solis und Wets, 1981) stellt eine simple Zufallssuche im Parameterraum dar. Zur Generierung von Punkten werden für diskrete und numerische Parameter einfach aus der diskreten bzw. stetigen Gleichverteilung Zufallszahlen gezogen. Die RS ist aufgrund ihrer Einfachheit, Flexibilität und Parallelität

sehr beliebt. So kann die RS im Gegensatz zur Gittersuche beispielsweise problemlos zu beliebigen Zeitpunkten abgebrochen oder erweitert werden. Bergstra und Bengio (2012) demonstrieren zudem eine Überlegenheit gegenüber der Gittersuche bezüglich der Güte gefundener Lösungen und empfehlen die Anwendung in der Hyperparameteroptimierung Maschineller Lernverfahren.

Referenzmodelle Als Referenz werden sämtliche für diese Domäne etablierten Modelle herangezogen, welche sich in diversen kleineren Benchmarkstudien zumindest teilweise als kompetitiv herausgestellt haben (Kammers u. a., 2011; Bøvelstad u. a., 2007; Van Wieringen u. a., 2009):

CoxPH: CoxPH-Modell basierend auf klinischen Daten sowie einer Menge von Genen, welche sich in Kratz u. a. (2012) als klinisch relevant herausgestellt haben. Diese Vorauswahl, im Folgenden als Kratz-Gene bezeichnet, setzt sich aus 19 Probe Sets zusammen, welche spezifisch für 17 Gene sind. Die Anzahl der Gene muss auf die Kratz-Gene beschränkt werden, da das CoxPH-Modell sich lediglich für „$p < n$"-Probleme eignet. Das Modell ist implementiert durch `survival::coxph` (Therneau und Grambsch, 2000).

randomForest: Random Survival Forest auf klinischen Daten und vorausgewählten Genen. Die Anzahl der Gene musste erneut auf die Kratz-Gene reduziert werden, da die Implementierung in `randomForestSRC::rfsrc` (Ishwaran u. a., 2008) für hochdimensionale Daten zu große Speicheranforderungen aufweist.

L1: CoxPH-Modell mit L_1-Regularisierung auf klinischen und genetischen Daten. Der Grad der Regularisierung wird durch ein Gradientenabstiegsverfahren automatisch bestimmt. Das Modell ist implementiert durch `glmnet::cv.glmnet` mit `alpha = 1` (Friedman, Hastie und Tibshirani, 2010; Simon u. a., 2011).

L2: CoxPH-Modell mit L_2-Regularisierung auf klinischen und genetischen Daten. Der Grad der Regularisierung wird durch ein Gradientenabstiegsverfahren automatisch bestimmt. Das Modell ist implementiert durch `glmnet::cv.glmnet` mit `alpha = 0` (Friedman, Hastie und Tibshirani, 2010; Simon u. a., 2011).

mboost: CoxPH-Modell mit Gradient Boosting auf klinischen und genetischen Daten. Die optimale Anzahl der Boosting-Iterationen wird automatisch mittels einer internen Kreuzvalidierung bestimmt, die technische Obergrenze beträgt $m = 500$. Das Modell ist implementiert durch `mboost::glmboost` (Hothorn u. a., 2010).

CoxBoost: CoxPH-Modell mit Likelihood Boosting auf klinischen und genetischen Daten. Die optimale Anzahl der Boosting-Iterationen wird automatisch mittels einer internen Kreuzvalidierung bestimmt. Das Modell ist implementiert durch `CoxBoost::optimCoxBoostPenalty` (Binder und Schumacher, 2008b).

Um eine gute Vergleichbarkeit zu gewährleisten, werden die Modelle natürlich auf denselben Aufteilungen der äußeren Kreuzvalidierung wie beim MBO-Tuning gebildet.

Der Vergleich mit Referenzmodellen ist jedoch nicht ganz fair, da sie die gesamte Masse der äußeren Trainingsaufteilung für ihr internes Tuning nutzen. Ein Abwägung, welches der Referenzmodelle für gegebene Daten zu wählen ist, ist so aber nicht möglich. Ein unverzerrtes Maß für die prädiktive Güte ist ebenfalls nicht verfügbar. Dieser Ansatz entspricht also der Strategie, a-priori eine bestimmte Modellklasse zu wählen, die Resultate als gegeben zu akzeptieren und auch dann nicht zu einem anderen Referenzmodell zu wechseln, falls das präferierte Modell unterdurchschnittlich abschneiden sollte.

Benchmark-Optimierung In der Praxis wird oft eine Art manuelle Optimierung über die zuvor beschriebenen sich selbst optimierenden Referenzmodelle durchgeführt, indem ein einfaches Benchmarking mittels innerer Kreuzvalidierung betrachtet wird. Das Referenzmodell mit bester durchschnittlicher Güte wird anschließend unter erneuter Verwendung des jeweiligen automatischen Tunings auf den gesamten Trainingsdaten der äußeren Kreuzvalidierung angepasst, um die finale Konfiguration zu bestimmen. Pro Datensatz wird also eines der Referenzmodelle basierend auf reduzierten Datenvolumen (identisches Volumen wie beim MBO oder der RS) gewählt. Dabei ist zu beachten, dass diese Art Benchmark-Optimierung (im Folgenden BenOpt abgekürzt) nur bei einer begrenzten Zahl diskreter Entscheidungen, wie hier der Wahl des sich selbst optimierenden Modells, möglich ist. Müssen dagegen stetige Parameter ebenfalls Berücksichtigung finden, so ist ein alternatives Vorgehen zur Optimierung wie etwa die MBO unumgänglich.

6.2.2 Parameterraum

Tabelle 6.2 gibt einen Überblick über den Parameterraum. Die ersten beiden Verfahren aus den Paketen `glmnet` und `penalized` stellen unterschiedliche Implementierungen von regularisierten CoxPH-Modellen bereit. Ein Likelihood-basiertes Boosting des CoxPH-Modells ist im Paket `CoxBoost` verfügbar, ein komponentenweises Boosting für generalisierte Lineare Modelle im Paket `mboost`. Im Paket `randomForestSRC` finden sich

Lerner	Werte	Erklärung
`glmnet::cv.glmnet` (CoxPH mit Elastischem Netz)		
`alpha`	$[0, 1]$	L_1/L_2 Verhältnis
`penalized::penalized` (CoxPH mit Elastischem Netz)		
`lambda1`	$[0, 200]$	L_1 Regularisierung
`lambda2`	$[0, 200]$	L_2 Regularisierung
`CoxBoost::CoxBoost` (Likelihood-basiertes Boosting)		
`stepno`	$\{1, \ldots, 500]$	Iterationen Boosting
`penalty`	$[0.001, 1000]$	Regularisierung
`stepsize.factor`	$[0.01, 2]$	Schrittweitenfaktor
`sf.scheme`	{`linear`,`sigmoid`}	Art Schrittweitenänderung
`mboost::glmboost` (Komponentenweises Boosting)		
`mstop`	$\{1, \ldots, 500\}$	Iterationen Boosting
`family`	{`CoxPh`,`Weibull`}	Verteilungsfamilie
`nu`	$[0, 1]$	Schrittweitenfaktor
`randomForestSRC::rfsrc` (Random Survival Forest)		
`ntree`	$\{1, \ldots, 1000\}$	Anzahl Bäume
`mtry.ratio`	$[0, 1]$	Anteil Variablen pro Baum
`nodesize`	$\{0, \ldots, 10\}$	Minimalgröße Terminalknoten
`splitrule`	{`lograng`,`logrankscore`}	Regel zur Aufteilung

Tabelle 6.2: Überblick über den aufgespannten Parameterraum der Modelle. Angegeben ist die Funktion, welche zur Modellierung benutzt wird. Einige Parameter tauchen in der zugehörigen angegebenen Funktionssignatur nicht auf, sind aber über Kontrollobjekte (siehe etwa die Dokumentation zu `mboost::boost_control()`) oder globale Optionen variierbar.

Überlebenszeitwälder. Einfache CoxPH-Modelle sind implizit durch die regularisierten CoxPH-Modelle enthalten, falls der zugehörige Strafterm als $\lambda = 0$ gewählt wird. Einzelne Überlebenszeitbäume sind durch Überlebenszeitwälder mit nur einem Baum ebenso abgedeckt.

Viele dieser Verfahren bieten ein automatisches Tuning ihrer wichtigsten Parameter an, beispielsweise etwa des Regularisierungsterms λ. Oft ist das Tuning jedoch recht rudimentär und geht zumeist nicht über eine ineffiziente kreuzvalidierte Vorwärts- oder Rückwärtssuche hinaus. Dies steht jedoch im Widerspruch zu der Idee des Modellmultiplexers, nämlich dass die Optimierung unterlegener Modelle zu vermeiden ist. Falls möglich und sinnvoll sind solche automatischen Optimierungen entsprechend deaktiviert. Eine Ausnahme stellt hier das elastische Netz aus dem Paket `glmnet` dar: Hier wird intern ein

effizientes Koordinatenabstiegsverfahren zur Optimierung verwendet (Friedman, Hastie und Tibshirani, 2010). Da ein externes Tuning hier viele Ressourcen verschenken würde, wurde das Tuning in diesem Fall nicht deaktiviert.

Eine Übersicht über die verwendeten vorgeschalteten Filter gibt Tabelle 6.3. Im Wesentli-

Filter	Parameter	Erklärung
Klinisch	–	Lässt klinische Kovariablen passieren
Kratz	–	Lässt klinisch Kovariablen und Kratz-Gene passieren
Klinisch+Kratz	–	Vereinigung beider vorheriger Filter
Varianz	%	Bewertung durch Varianz-Score
Score	%	Bewertung durch χ^2-Score des CoxPH-Modells
C-Index (Train)	%	Bewertung durch C-Index (Trainingsmenge)
C-Index (Test)	%	Bewertung durch C-Index (3x-Kreuzvalidiert)
mRMR	%	Bewertung durch mRMR (C-Index / Korrelation)

Tabelle 6.3: Überblick über den aufgespannten Parameterraum der Filter.

chen lassen sich die Filter in zwei Gruppen aufteilen: Die ersten drei Filter („Klinisch“, „Kratz“ und „Kratz+Klinisch“) lassen nur klinische Kovariablen zu oder vergleichen die Namen der Kovariablen mit Listen von erlaubten Genen, sind also Literaturfilter (siehe Unterabschnitt 3.3.3). Sie agieren damit datenunabhängig und sind frei von Parametern. Die übrigen fünf Filter bewerten jedes Gen und verteilen einen Score, wobei sich die Anzahl der Gene, welche dem Überlebenszeitmodell übergeben werden sollen, durch eine Prozentzahl regeln lässt. Per Definition ist für jeden dieser Filter ein hoher Score als positiv bzw. relevant zu verstehen.

6.2.3 Surrogat Modell

Als Surrogat-Modell in der Modell-basierten Optimierung kommt ein Wald aus Regressionsbäumen (*random regression forest*) zum Einsatz. Die Funktionsweise von Regressionswäldern ist zusammen mit den Überlebenszeitwäldern in Abschnitt 3.5 beschrieben worden. Für den gegebenen Versuchsaufbau spricht vieles für den Einsatz von Wäldern für die Surrogatregression:

- Regressionswälder operieren problemlos gleichzeitig auf diskreten, ordinalen und stetigen Variablen. Da der Parameterraum gemischt skaliert ist, ist dies eine notwendige Eigenschaft.

- Es existieren gut bekannte und verständliche Strategien zur Imputation fehlender Werte. Durch die bereits erläuterten Abhängigkeiten der Parameter kommt es systematisch zu fehlenden Werten. Als Imputation für diskrete Variablen wird daher eine neue Stufe `"<NA>"` eingeführt, welches nicht mit dem von `R` unterstützen `NA` zu verwechseln ist, sondern eine eigens definierte Faktorstufe darstellt. Der Regressionswald kann dieses so problemlos erkennen und falls nötig speziell behandeln. Da der Regressionswald zudem robust gegenüber Ausreißern ist, stellt die Imputation durch einen sehr kleinen oder sehr großen Wert eine sinnvolle Strategie für stetige Variablen dar. Der einzelne Baum kann hier entsprechend die fehlenden von den nicht-fehlenden Beobachtungen mit einer einfachen Regel trennen und in den folgenden Knoten gegebenenfalls gezielt weiter analysieren.

- Eine anschließende Analyse des Waldes und seiner *Variable Importance* ermöglicht eine ausführliche Analyse des Parameterraums. Dadurch kann beispielsweise genauer untersucht werden, ob die Wahl des Filters oder die Wahl des Modells wichtiger ist.

Es wird die Implementierung des Regressionswaldes aus dem Paket `randomForest` (Liaw und Wiener, 2002) verwendet. Dabei wurden im Wesentlichen die Standardeinstellungen übernommen, lediglich die Anzahl der Bäume wurde reduziert. Das bedeutet konkret, dass in jedem Schritt ein Wald mit 100 Regressionsbäumen angepasst wird. Jeder Baum wird auf einer Boostrap-Stichprobe desselben Umfangs wie die Trainingsdaten angepasst. Dabei wird jeweils lediglich ein Drittel der verfügbaren Kovariablen verwendet. Die minimale Größe von Terminalknoten wurde mit fünf Beobachtungen bei dem Standardwert für Regressionsanalysen belassen.

Der Regressionswald wurde durch einen `mlr`-Wrapper um eine Imputationsstrategie erweitert: Für diskrete Parameter wird wie beschrieben eine neue Stufe zur Indikation fehlender Werte eingeführt (`"<NA>"`). Fehlende Werte in stetigen Parametern werden währenddessen durch einen Wert ersetzt, welcher die doppelte beobachtete Spannweite des beobachteten Maximums entfernt liegt, also

$$\max(x) + 2\left(\max(x) - \min(x)\right).$$

Damit wird eine sehr generische Imputation benutzt, welche keinerlei Informationen aus den Testdaten verwendet und somit nicht Gefahr läuft, an dieser Stelle unnötig Überoptimismus in das Experiment zu bringen.

Zur Varianzschätzung wird ein Bootstrap-Schätzer verwendet, vergleiche auch die Diskussion in Abschnitt 4.2. Das bedeutet, dass zur Schätzung der Varianz fünf weitere Zufallsregressionswälder angepasst werden, jeweils auf Bootstrap-Stichproben der Trainingsdaten. Der Umfang der Stichprobe entspricht dabei dem Umfang der Trainingsdaten, sie deckt entsprechend circa 63.2 % der Beobachtungen ab.

6.2.4 MBO

Es sind zwei Varianten der Modell-basierten Optimierung mit dem beschriebenen Regressionswald als Surrogatmodell durchgeführt worden.

Gemeinsamkeiten Beide Varianten erhalten ein Budget von jeweils 3000 Punktevaluierungen, durch die dreifache Kreuzvalidierung also 9000 Gesamtevaluierungen. Die ersten 100 Evaluierungen werden für das initiale Design als *Latin Hypercube* (Stein, 1987) verbraucht. Es verbleiben entsprechend 2900 Evaluierungen für die iterative Optimierung. Das LCB-Kriterium wird zur Bewertung interessanter Regionen benutzt und durch die Fokussuche optimiert. Die Fokussuche wird dreimal neu gestartet, jeweils mit fünffacher Fokussierung und einem Raster aus 10 000 Punkten.

Variante MBOK (Klassisch) Diese Variante wertet in jeder Iteration genau den Punkt aus, welcher bezüglich Infill-Kriterium ideal erscheint. Dies ist analog zu dem in Abschnitt 4.2 skizzierten Vorgehen des Algorithmus. Jedoch läuft dieses Verfahren durch die Varianzschätzung des Regressionswaldes Gefahr, unter gewissen Bedingungen nicht ausreichend zu explorieren. Dies liegt an der anderen Varianzschätzung. Beim Kriging-Modell besteht ein direkter Zusammenhang zwischen der geschätzten lokalen Unsicherheit und der Distanz zur nächsten Evaluation, bei dem Bootstrap-Varianzschätzer des Waldes ist dies dagegen nicht der Fall. Dies hat zur Folge, dass die Varianz beispielsweise an Randbereichen, an denen noch keine oder nur sehr wenige Evaluationen erfolgt sind, sehr gering geschätzt werden kann, da hier alle Boostrap-Wälder übereinstimmen. Folgend werden diese Bereiche nicht als interessant erkannt und auch in späteren Iterationen nicht ausreichend exploriert.

Variante MBOA (Alternierend) In dieser Variante wird, wie in Thornton u. a. (2012) vorgeschlagen, nur jeder zweite Punkt durch das Infill-Kriterium bestimmt. Der jeweils

andere Punkt wird zufällig gewählt. Jeder zweite Schritt ist somit eine forcierte zufällige Exploration, um das Problem der Varianzschätzung zu mindern. Zwar kann der Infill-Schritt stets noch explorieren, insbesondere dann wenn sich zufällig ein großer Wert für den Balancierungsterm λ_{LCB} realisiert, insgesamt tendiert dieser Schritt aber vergleichsweise eher zur Ausschöpfung bereits besuchter, aber interessanter Regionen.

6.3 Auswertung

Die resultierenden Hyperparameter-Konfigurationen der beiden MBO-Varianten und der Random Search werden zunächst im folgenden Unterabschnitt 6.3.1 isoliert betrachtet und auf Auffälligkeiten untersucht. In dem daran anschließenden Unterabschnitt 6.3.2 werden die einzelnen automatisch getunten Referenzmodellen näher betrachtet. In Abschnitt 6.3.3 erfolgt ein Vergleich mit der Benchmark-Optimierung. Abschließend werden in Abschnitt 6.3.4 alle Ansätze gemeinsam betrachtet, verglichen und bewertet.

Die zur Auswertung eingesetzte `R`-Version sowie die Versionen direkt verwendeter Pakete sind in Tabelle A.1 im Anhang A aufgelistet.

6.3.1 Untersuchung der MBO-Varianten

Die Optimierung wurde auf dem Hochperformanz-Cluster LiDOng der TU Dortmund durchgeführt. Jeder der fünf MBO-Läufe auf den sieben Datensätzen ist als ein Experiment in `BatchExperiments` definiert worden. Innerhalb dieses Experiments wurde mittels `parallelMap` die dreifache Kreuzvalidierung zur Punktevaluation parallelisiert, entsprechend reserviert jeder Job drei Cores auf einem Node. Dies entspricht einer maximalen parallelen Auslastung von $5 \times 7 \times 3 = 105$ Kernen. Der Speicherverbrauch wurde relativ konservativ gewählt, so dass jeder der 40 Jobs insgesamt 12 GB für die drei Threads reserviert. Die maximale Laufzeit (Walltime) der Experimente wurde mit maximal acht Stunden bemessen. Auf dem LiDOng-System bewirken diese Einstellungen eine Einteilung in die Warteschlange „med_quad“, einer Warteschlange für speicherintensive Jobs mit mittlerer Laufzeit. Dies entspricht einem Kompromiss, da hier einerseits ein hoher Grad an Parallelität möglich ist, anderseits aber die Jobs alle acht Stunden durch den Scheduler automatisch terminiert werden. Um den Datenverlust durch die Programmabbrüche zu verhindern, wird nach jeder Punktevaluation der aktuelle Zustand der Optimierung (als Optimierungspfad, siehe `?OptPath` in `ParamHelpers`) persistent auf

die Festplatte gespeichert. Ein erneut gestarteter Job kontrolliert entsprechend zunächst, ob eine Datei existiert, mit der vorangegangene Berechnungen fortgeführt werden können. Diese Datei ist entweder ein erfolgreich gespeicherter Optimierungspfad, oder ein Backup des davor gespeicherten Optimierungspfades. Denn wenn der Scheduler den Prozess während des Schreibzugriffs terminiert, sind korrumpierte Dateien unausweichlich. Durch diesen Mechanismus geht also maximal eine einzige Punktevaluation verloren. Dieses Wiederaufnahme-Verfahren benötigt außerdem eine Art Dienst auf dem Master-Knoten, welcher in regelmäßigen Abständen abgelaufene Jobs identifiziert und diese erneut dem Scheduler zur Einsortierung in die Warteschlange übergibt. Die Parallelisierung der Random Search ist dagegen trivial. Zur besseren Vergleichbarkeit wurden jedoch ebenfalls dieselben 100 Punkte mittels LHS in den Raum gelegt wie bei der MBO, die folgenden 2900 Punkte aber entsprechend zufällig bestimmt.

Beide MBO-Varianten sowie die Random Search wurde auf den fünf Trainingsaufteilungen der äußeren Kreuzvalidierung angewendet, wobei jede der 3000 Punktevaluationen auf einer inneren dreifachen Kreuzvalidierung basiert. Das Budget umfasst also jeweils $B = 9000$ Evaluationen. Im Anschluss wird die Modellkonfiguration gewählt, für welche der beste Konkordanzindex C beobachtet werden konnte. Diese Parametrisierung wird benutzt, um ein Modell auf allen Trainingsdaten der jeweiligen äußeren Kreuzvalidierung zu bilden. Die Vorhersage auf den Testdaten der äußeren Kreuzvalidierung wird berechnet und mittels des Konkordanzindizes evaluiert, was in Abbildung 6.3 visualisiert ist. Die resultierenden Konfigurationen, also welche Modelle, Filter und zugehörige Hyperparameter tatsächlich am Ende der Optimierung gewählt wurden, sind in tabellarischer Form in den Tabellen A.2 bis A.8 im Anhang A zu finden. Eine zusätzliche Übersicht über die selektierten Modelle und Filter der klassischen Variante ist durch Abbildung B.1 und B.2 gegeben.

Augenscheinlich liegen beide MBO-Varianten in etwa gleichauf, der Zufallsschritt in Variante MBOA scheint keinen gravierenden Einfluss zu haben. Dies ist für Variante MBOK gleichzeitig positiv und negativ zu bewerten: Einerseits scheint die zufällige Wahl von λ_{LCB} für ausreichend Exploration zu sorgen, andererseits konnte dann aber die Exploitation trotz doppeltem Budget nicht richtig greifen. Eine mögliche Erklärung liefert die Höhe des Budgets, eine verschwenderische Einstellung kann eventuelle Unterschiede verdecken. Daher wurde zunächst der Einfluss des Budgets analysiert. In Abbildung B.3 ist das gewährte Budget gegen die Nummer des Iterationsschrittes korrespondierend zum finalen Konfigurationsvorschlag abgetragen, also ab welchem Iterationsschritt ohne Änderung des finalen Vorschlages hätte abgebrochen werden können. Die Linien verbinden dabei die

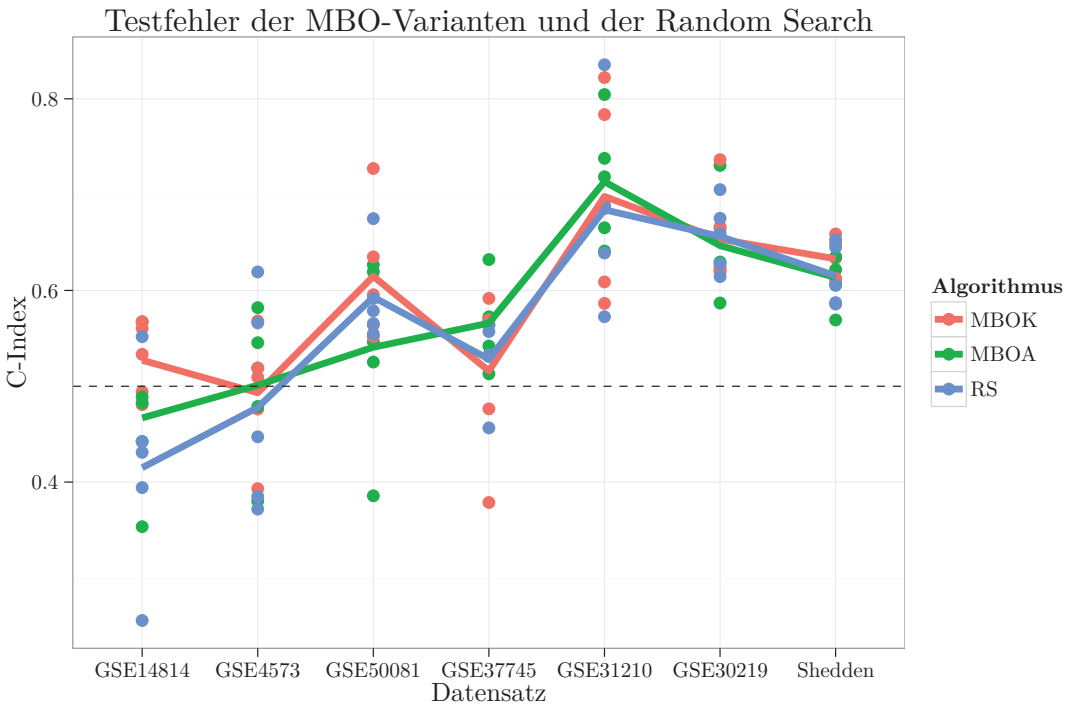

Abbildung 6.3: Testfehler (Konkordanzindex, y-Achse) auf den sieben Datensätzen (x-Achse), von links nach rechts aufsteigend sortiert nach Anzahl Beobachtungen. Linien verbinden arithmetische Mittel über die Aufteilungen der jeweiligen fünffachen Kreuzvalidierungen. Die gestrichelte Linie repräsentiert die Zufallsgrenze $C = 0.5$.

arithmetischen Mittel der fünffachen Kreuzvalidierungen, wobei ihr monotones Steigen auf einen kontinuierlichen Progress in der Optimierung schließen lässt. Der Optimierungsfehler, jeweils im oberen Teil der Abbildungen B.4 (Klassische Variante MBOK) und B.5 (Alternierende Variante MBOA), lassen ebenfalls eine fortschreitende Verbesserung vermuten, wobei die erzielten Verbesserungen eher gering sind. Allerdings wären auch kleine Verbesserungen auf den hochdimensionalen Daten wichtig. Bei Betrachtung der Testfehler, zu finden in den beiden Abbildungen B.4 und B.5 im jeweils unteren Teil, wird jedoch deutlich, dass hier kein Progress in der wirklichen Vorhersagequalität vorhanden ist, sondern nur ein Progress im Optimierungsfehler. Dies gilt auch für die RS, deren Optimierungs- und Testfehler in Abbildung B.6 zu finden sind. Es werden also ab einem relativ geringen Budget nur Punkte gefunden, welche auf der inneren Trainingsmenge interessanter sind, aber nicht zur Vorhersagegüte auf der äußeren Testmenge beitragen.

Auch im Vergleich mit der Random Search kann sich optisch keine der beiden MBO-

Varianten über die Datensätze hinweg deutlich absetzen. Bei genauer Betrachtung kann jedoch MBOK gegenüber der RS favorisiert werden, lediglich auf dem Datensatz GSE37745 schneidet die RS besser ab, und dies nicht übermäßig deutlich. Der Vergleich mit MBOA ergibt ein gemischtes Bild. Da es sich bei der MBOA-Variante um einen Mittelweg aus Modell-basiertem Vorgehen und zufälliger Suche handelt, wären eigentlich auch die Resultate zwischen MBOK und RS zu erwarten. Dies ist aus Grafik 6.3 jedoch nicht direkt erkennbar. Lediglich nach weiterer Aggregation der Konkordanzindizes lässt sich eine Aussage treffen. Dazu wird zunächst pro Datensatz und Algorithmus der gemittelte Konkordanzindex über die fünffache Kreuzvalidierung berechnet. Pro Datensatz werden Rangstatistiken für die drei Optimierungsstrategien gebildet und diese anschließend erneut mit dem arithmetischen Mittel zusammen gefasst. Die gemittelten Ränge der gemittelten Konkordanzindizes zeigen auf, dass die klassische MBO-Variante MBOK mit einem durchschnittlichen Rang von $r = 1.71$ im Vergleich mit MBOA mit $r = 2$ und der RS mit $r = 2.29$ am besten abgeschnitten hat.

Warum die Unterschiede zwischen den Optimierungsverfahren allerdings so gering sind, lässt sich an dieser Stelle nicht eindeutig klären. Plausibel erscheint, dass die Güte innerhalb guter Regionen des Hyperparameterraums, etwa zu einem bestimmten Überlebenszeitmodell, so homogen ist, dass jegliche Unterschiede in den zugehörigen Modellparametern durch die Zufälligkeit der Trainingsaufteilungen verrauscht werden. Daraus ließe sich schließen, dass Regionen nur sehr grob besucht werden müssen, um kompetitive Konfigurationen zu generieren. Eine andere Erklärung wäre, dass die drei Verfahren tatsächlich in etwa gleich gut bzw. schlecht sind, dies würde dann im anschließenden Unterabschnitt beim Vergleich mit den Referenzmodellen deutlich werden.

6.3.2 Vergleich mit Referenzmodellen

Die Referenzmodelle wurden auf den gleichen Aufteilungen der äußeren Kreuzvalidierung wie die Optimierer untersucht. Ob und wie ein inneres Resampling zur Vermeidung einer Überanpassung durchgeführt wird, bleibt hier den entsprechenden automatischen Tunern der jeweiligen Implementierungen überlassen. Durch `mlr` ist lediglich sichergestellt, dass ausschließlich die Trainingsdaten der äußeren Aufteilung zum Lernen benutzt werden und die Evaluation auf den Testdaten stattfindet. Die einzelnen Modellanpassungen sind natürlich unabhängig, eine Parallelisierung hier entsprechend simpel. Jedoch versagte der automatische Tuning-Algorithmus der CoxBoost-Implementierung: Auch nach 48 Stunden Laufzeit war für zwei Aufteilungen kein Model verfügbar, während für andere Aufteilungen

des gleichen Datensatzes bereits nach einigen Minuten Ergebnisse vorhanden waren. In fünf Fällen beendete CoxBoost sich reproduzierbar nach einigen Stunden selbst mit einer Fehlermeldung. Die Likelihood-basierten Boosting-Modelle konvergieren also nicht immer, hier wurde gegebenenfalls ein Konkordanzindex von $C = 0.5$ (konstante bzw. zufällige Prognosen) imputiert.

Die gemittelten Konkordanzindizes der Referenzmodelle sind in Tabelle 6.4 aufgeführt. Dabei ist auffällig, dass trotz automatischen Tunings nicht immer Ergebnisse erzielt werden

Datensatz	Beobachtungen	Ereignisse	CoxPH		CoxBoost	
GSE14814	90	38	0.572*	(0.141)	0.487	(0.113)
GSE4573	129	67	0.472	(0.106)	0.440	(0.064)
GSE50081	181	75	0.614*	(0.078)	0.591	(0.058)
GSE37745	194	143	0.534	(0.028)	0.545	(0.043)
GSE31210	226	35	0.603	(0.135)	0.727	(0.053)
GSE30219	268	170	0.629	(0.045)	0.641	(0.056)
Shedden	442	236	0.636	(0.024)	0.648	(0.029)

Datensatz	L1		L2		mboost		randomForest	
GSE14814	0.500	(0.000)	0.415	(0.138)	0.487	(0.125)	0.492	(0.084)
GSE4573	0.500	(0.000)	0.509*	(0.187)	0.429	(0.067)	0.480	(0.041)
GSE50081	0.500	(0.000)	0.591	(0.050)	0.590	(0.061)	0.605	(0.047)
GSE37745	0.500	(0.000)	0.538	(0.061)	0.524	(0.051)	0.579*	(0.059)
GSE31210	0.598	(0.099)	0.753*	(0.041)	0.718	(0.055)	0.677	(0.143)
GSE30219	0.500	(0.000)	0.610	(0.064)	0.642*	(0.054)	0.632	(0.044)
Shedden	0.578	(0.072)	0.602	(0.047)	0.650*	(0.020)	0.616	(0.037)

Tabelle 6.4: Arithmetisches Mittel und Standardabweichung der Konkordanzindizes, evaluiert mittels fünffacher Kreuzvalidierung. Das beste Verfahren pro Datensatz ist mit einem Stern markiert. Die Datensätze sind sortiert nach Anzahl der Beobachtungen (Patienten). Die Anzahl der Ereignisse (Tode) sind ebenfalls aufgeführt.

können, welche sich von zufälligen Ergebnissen absetzen können. Das veranlasst einerseits Sorge über die Datenqualität, zeigt anderseits aber auch auf, dass eine unbedachte Anwendung eines favorisierten Modells zu schwerwiegenden Fehlentscheidungen führen kann. Auf einigen Datensätzen wie GSE50081 scheint die prädiktive Güte primär aus den klinischen Kovariablen gewonnen zu werden – das niedrigdimensionale CoxPH-Modell schneidet hier am besten ab – auf anderen Datensätzen wie GSE31210 kann scheinbar aus den genetischen Kovariablen viel wertvolle Information extrahiert werden. Auch die

Analyse der selektierten Filter offenbart wertvolle Eigenschaften der Datensätze: Auf dem Datensatz Shedden wird primär ein Filter gewählt, welcher alle genetischen Variablen außer die Kratz-Gene eliminiert (vergleiche Abbildung B.2). Solches Wissen lässt sich indessen natürlich nur dann erlangen, wenn mehrere Modelle betrachtet und miteinander verglichen werden. Deshalb erscheint die Strategie, an einer bestimmten Modellklasse festzuhalten, eine schlechte Wahl zu sein. Die Ergebnisse aus Tabelle 6.4 sind ebenfalls zusammen mit den Resultaten der MBOK-Variante in Abbildung 6.4 dargestellt. Es

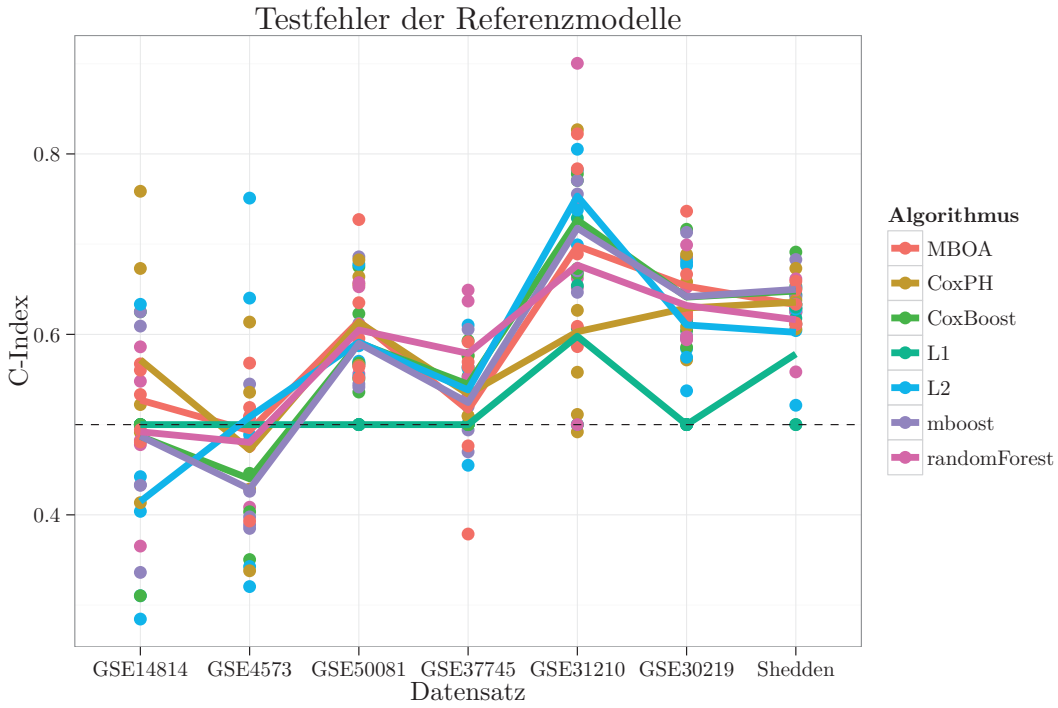

Abbildung 6.4: Testfehler (Konkordanzindex, y-Achse) der Referenzmodelle auf den sieben Datensätzen (x-Achse), von links nach rechts aufsteigend sortiert nach Anzahl Beobachtungen. Linien verbinden arithmetische Mittel über die Aufteilungen der jeweiligen fünffachen Kreuzvalidierungen. Zur besseren Vergleichbarkeit sind die Resultate der klassischen MBO-Variante MBOK ebenfalls eingezeichnet. Die gestrichelte Linie repräsentiert die Zufallsgrenze $C = 0.5$.

ergibt sich ein ähnliches Bild wie bei dem Vergleich der Optimierer untereinander. Die Wahl des Modells scheint einen geringeren Einfluss als die Aufteilung des Resamplings zu haben. Eine Ausnahme stellt hier lediglich die Lasso-Regression dar, welche deutlich schlechter als die anderen Referenzmodelle abschneidet. Ein gemittelter Konkordanzindex von $C = 0.5$ in Tabelle 6.4 zeigt für die vier kleinsten Datensätze an, dass hier auf ein

Null-Modell, also ein Modell ohne Kovariablen geschrumpft worden ist.

Auch über die Datensätze lassen sich erneut einige Aussagen aus Abbildung 6.4 ableiten. Auf den ersten beiden Datensätzen greift quasi kein Modell. Dies könnte entweder an der geringen Stichprobenanzahl liegen (die Datensätze sind von links nach rechts nach Anzahl der Beobachtungen sortiert, siehe auch Tabelle 6.4), oder auf ein Problem in der Erhebung oder Vorverarbeitung deuten. Ein reiner Effekt der Stichprobengröße scheint jedoch nicht vorzuliegen, denn dann wäre eine monotone Steigung zu beobachten. Der Datensatz GSE31210 scheint erneut qualitativ am besten, zumindest gelingt es den Modellen hier am deutlichsten, sich von der Zufallsgrenze $C = 0.5$ abzusetzen. Dieser Datensatz beinhaltet andererseits jedoch auch ausschließlich Patientinnen und Patienten mit Adenokarzinomen (vergleiche Tabelle A.10). Dies könnte darauf hindeuten, dass bei der Modellierung auf den heterogeneren Datensätzen die Flexibilität fehlt, die Unterschiede zwischen den Subgruppen verschiedener Tumortypen angemessen auszugleichen. Alternativ kann argumentiert werden, dass die Flexibilität zwar vorhanden sei, aber der effektive Stichprobenumfang pro Tumortyp wesentlich geringer ausfällt. Dem widersprechen allerdings die etwas schlechteren Ergebnisse der Kohorte Shedden, welcher sich ebenfalls nur aus Adenokarzinomen zusammensetzt, aber über fast doppelt so viele Beobachtungen wie der Datensatz GSE31210 verfügt. Vermutlich ist hier eine Mischung vieler Faktoren dafür verantwortlich, ob ein Datensatz gut lernbar ist oder nicht. Eine Subgruppenanalyse könnte hier mehr Klarheit schaffen.

Ein Vergleich der Referenzmodelle über gemittelte Rangstatistiken, wie er bei den Optimierern untereinander erfolgt ist, findet sich im Unterabschnitt 6.3.4 im Rahmen eines globalen Vergleichs aller Analysestrategien.

6.3.3 Vergleich mit Benchmarking-Optimierung

Die Benchmark-Optimierung wurde zur Gewährleistung der Vergleichbarkeit erneut auf identischen Aufteilungen der äußeren Kreuzvalidierung durchgeführt. Auf der Trainingsmenge der äußeren Aufteilung wurde eine dreifache Kreuzvalidierung gewählt, um eines der Referenzmodelle auszuwählen. Dieses wurde dann auf den kompletten Trainingsdaten angepasst, um die Vorhersagen für die äußere Testmenge zu generieren. Das Resultat ist in Abbildung 6.5 dargestellt. Am häufigsten wurde dabei das Likelihood-basierte Boosting-Modell CoxBoost gewählt, in insgesamt 12/35 Fällen. Am zweithäufigsten fiel die Wahl auf ein Ridge-Modell (8/12). Das niedrigdimensionale CoxPH-Modell sowie das

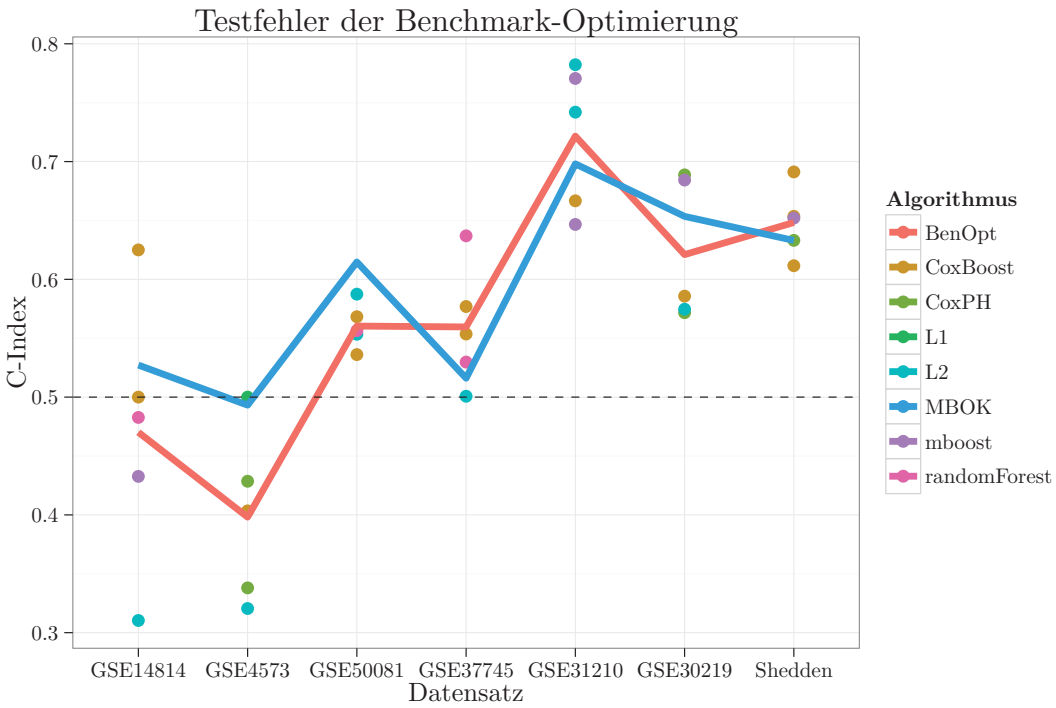

Abbildung 6.5: Testfehler (Konkordanzindizes, y-Achse) der durch BenOpt selektierten Referenzmodelle auf den sieben Datensätzen (x-Achse), von links nach rechts aufsteigend sortiert nach Anzahl Beobachtungen. Die Linie zu BenOpt verbindet arithmetische Mittelwerte der Konkordanzindizes über die fünffache Kreuzvalidierung. Zur besseren Vergleichbarkeit sind die Resultate der klassischen MBO-Variante MBOK ebenfalls eingezeichnet. Die gestrichelte Linie repräsentiert die Zufallsgrenze $C = 0.5$.

Gradienten-basierte Boosting wurden in fünf Fällen selektiert, der Überlebenszeitwald viermal und die Lasso-Regression in einer einzigen Aufteilung der Kreuzvalidierungen. Im Vergleich mit MBOK ist keine gleichmäßige Überlegenheit einer Optimierungsstrategie über die Datensätze hinweg auszumachen. Damit scheint die Benchmark-Optimierung eine einfache Alternative zu MBO zu sein. Ein Vergleich über aggregierte Rangstatistiken im direkt folgenden Unterabschnitt 6.3.4 widerlegt dies jedoch. Es bleibt außerdem zu bedenken, dass der rechnerische Aufwand für die BenOpt mit der Anzahl der Modelle skaliert. Dies ist darin begründet, dass bei der Benchmark-Optimierung jedes Modell einzeln getunt werden muss. MBO kann sich dagegen durch die Verwendung des Multiplexers auf interessante Modelle konzentrieren. Ist also abzusehen, dass die Lasso-Regression schlecht abschneidet, kann die dadurch verfügbar gewordene Rechenzeit für andere Modelle genutzt werden. Außerdem ist die BenOpt nicht für solche Modelle anwendbar, welche

über kein automatisches Tuning ihrer numerischen Parameter verfügen. Dies schließt insbesondere auch kombinierte Methoden, wie die vorgeschaltete Variablenselektion beim Modellmultiplexer, aus.

6.3.4 Gemeinsame Betrachtung

Da die visuelle Untersuchung der Optimierungsverfahren kein eindeutiges Ergebnis liefert, erfolgt nun eine Analyse zusammengefasster Maßzahlen. In Tabelle 6.5 finden sich die Konkordanzindizes der jeweiligen Optimierungsverfahren, aufgeschlüsselt nach Datensatz und gemittelt über die fünffache äußere Kreuzvalidierung.

Datensatz	MBOK	MBOA	RS	BenOpt
GSE14814	0.527* (0.039)	0.467 (0.078)	0.415 (0.107)	0.470 (0.114)
GSE4573	0.493 (0.065)	0.501* (0.077)	0.478 (0.110)	0.398 (0.072)
GSE50081	0.615* (0.071)	0.541 (0.097)	0.593 (0.048)	0.560 (0.019)
GSE37745	0.516 (0.088)	0.566* (0.044)	0.528 (0.043)	0.560 (0.052)
GSE31210	0.698 (0.104)	0.713 (0.064)	0.684 (0.097)	0.722* (0.061)
GSE30219	0.653 (0.050)	0.647 (0.054)	0.656* (0.036)	0.621 (0.060)
Shedden	0.633 (0.021)	0.613 (0.027)	0.615 (0.032)	0.648* (0.029)

Tabelle 6.5: Arithmetisches Mittel und Standardabweichung der Konkordanzindizes der Optimierungsverfahren, evaluiert mittels fünffacher Kreuzvalidierung. Das beste Verfahren pro Datensatz ist mit einem Stern markiert. Datensätze sortiert nach Anzahl der Beobachtungen.

Auf den kleineren Datensätzen haben alle Optimierungsverfahren Probleme besser als ein zufälliger Prädiktor mit $C = 0.5$ zu sein. Dies gilt ebenso für die Referenzmodelle, aufgeführt in Tabelle 6.4. Für den kleinsten Datensatz GSE14814 erreichte das einfache CoxPH-Modell als einziger Ansatz mit $C = 0.527$ eine besser als zufällige prädiktive Güte. Hier scheint die Hinzunahme von genetischen Kovariablen demnach eher nachteilig zu sein. Auf dem zweitkleinsten Datensatz GSE4573 kann kein Verfahren etwas lernen, alle Konkordanzindizes sind im Schwankungsbereich zufälliger Prognosen oder zumeist sogar deutlich schlechter als zufällige Prognosen. Diese Kohorte scheint sehr heterogen zu sein, so dass die Verteilungen der Trainings- und Testmenge sich sehr oft unterscheiden und die Modelle in der Folge nicht generalisieren. Eine andere Erklärung könnte die Stichprobengröße liefern: Möglicherweise stellen 181 Beobachtungen mit 75 Ereignissen (also der drittkleinste Datensatz GSE50081) eine untere Schranke dar, ab der eine

Modellierung in diesem hochdimensionalen Raum möglich und sinnvoll ist. Auf den übrigen Datensätzen konnte von allen Optimierungsverfahren und Referenzmodellen, mit Ausnahme der L_1-Regression, etwas gelernt werden.

Die Information aus Tabelle 6.5 lassen sich hier, analog zur Berechnung in Unterabschnitt 6.3.1, mittels Rangstatistiken zum besseren Vergleich weiter aggregieren. Für jeden Datensatz wird also die Rangfolge der resultierenden gemittelten Testfehler aller Auswertungsstrategien, auch die der Referenzmodelle aus Tabelle 6.4, berechnet. Sollten zwei Auswertungsstrategien einen identischen Konkordanzindex besitzen (etwa $C = 0.5$), so werden gegebenenfalls Durchschnittsränge gebildet. Anschließend werden die Rangstatistiken mit dem arithmetischen Mittel gruppiert nach Auswertungsstrategie zusammengefasst. Dies ist in Tabelle 6.6 aufgeführt. Für die linke Tabelle wurden alle

Algorithmus	$\bar{r}_a$
MBOK	4.14
randomForest	4.71
CoxBoost	4.86
CoxPH	5.14
MBOA	5.29
mboost	5.43
L2	5.71
RS	5.86
BenOpt	5.86
L1	8.00

Algorithmus	$\bar{r}_c$
CoxBoost	4.00
MBOK	4.60
mboost	4.80
randomForest	4.80
BenOpt	4.80
RS	5.20
MBOA	5.40
CoxPH	5.60
L2	5.80
L1	10.00

Tabelle 6.6: Mittelwerte der Durchschnittsränge der jeweils 10 Auswertungsstrategien, aufsteigend sortiert von gut (hoher mittlerer Konkordanzindex) zu schlecht (niedriger mittlerer Konkordanzindex). Links: Berechnung auf allen Datensätzen. Rechts: Berechnung ohne Datensätze GSE14814 und GSE4573.

Datensätze verwendet. Die klassische MBO-Variante MBOK schneidet hier am besten ab, aber auch der Überlebenszeitwald, eine vergleichsweise eher selten benutzte Alternative zu den CoxPH-Modellen, erreicht eine sehr gute mittlere Rangstatistik. Die Lasso-Regression hat, wie es sich bereits durch Grafik 6.4 abgezeichnet hat, deutlich am schlechtesten abgeschnitten. Die Benchmark-Optimierung, der aktuelle de-facto Standard in der Praxis, ist auf dem vorletzten Platz und stellt damit eine schlechtere Alternative dar, als sich apriori auf eine beliebige Modellklasse außer der Regression mit Lasso-Regularisierung festzulegen.

Für die rechte Tabelle in 6.6 wurden die beiden Datensätze mit den wenigsten Beob-

achtungen, auf denen nicht ein einziges Verfahren funktioniert hat, ignoriert. Dadurch kommt das Referenzmodell CoxBoost, also das Likelihood-basierte Boosting, an die Spitze. Gleichzeitig rutscht das MBOK im Vergleich mit der linken Tabelle von dem ersten Rang auf den Zweiten. Die Benchmark-Optimierung rückt zusammen mit den Optimierern RS und MBOA ins Mittelfeld, Schlusslicht bilden die beiden mit L_1 bzw. L_2 regularisierten Modelle. Die beiden Boosting-basierten Modelle (Rang 1 und 3) besitzen beide eine bemerkenswerte prädiktive Güte, der Vergleich mit der linken Tabelle offenbart allerdings eine gewisse Neigung auch dort etwas lernen zu wollen, wo scheinbar keine Informationen sind. Sie tendieren in solch einem Szenario also eher als die anderen Ansätze zur Überanpassung. Ein Verhalten wie bei der Lasso-Regression, welches hier einfach ein Null-Modell liefert, wäre stattdessen an dieser Stelle wünschenswert.

Da normalerweise bei der Analyse eines neuen Datensatzes nicht bekannt ist, ob eine Methode überhaupt greift oder ob verwertbare Informationen in den Daten stecken, ist letztlich die linke Tabelle mit der Berücksichtigung aller Datensätze repräsentativer. Denn selbst wenn scheinbar keinerlei Information in den Daten steckt, sollte das Verfahren keine Prädiktionen liefern, welche deutlich schlechter als zufällig sind. Dies wird aber nur in der linken Tabelle entsprechend aufgenommen und gegebenenfalls bestraft.

Selbstverständlich sollten die Rangzahlen nicht überinterpretiert werden. Andererseits drückt das Ranking auf der linken Seite in Tabelle 6.6 aus, welche Strategie nach bestem Wissen für einen neuen Datensatz zu wählen ist. In diesem Fall ist dies also die Optimierung mittels MBOK. Wird dagegen ein starker Effekt der Stichprobengröße unterstellt, also dass auf den ersten beiden Datensätzen schlicht zu wenig Eingabe vorhanden ist, die einzelnen Methoden ansonsten aber tadellos funktionierten, so gewinnt die rechte Seite der Tabelle 6.6 an Bedeutung. Liegt also ein neuer Datensatz mit über 200 Beobachtungen vor, könnte die rechte Tabelle einen besseren, weil repräsentativeren Anhaltspunkt liefern. Allerdings hat das erstplatzierte CoxBoost das Problem, nicht immer zu konvergieren, was eine alleinige Anwendung in der Praxis unmöglich macht. Da zudem ein Stichprobeneffekt einerseits schwer bestimmbar ist und zusätzlich die Modell-basierte Optimierung mit dem zweiten Rang ebenfalls in den oberen Rängen vertreten ist, ist eine solche Abwägung vermutlich gar nicht notwendig. Abbildung B.1 zeigt zudem, dass MBOK im Wesentlichen zwischen den drei Modellklassen „CoxBoost“, „mboost“ und „randomForest“ springt. Dadurch besitzt MBO inherent mehr Flexibilität – sind also auf einem Datensatz Interaktionen von Kovariablen wichtig, kann MBO hier von einem Boosting-Modell auf einen Überlebenszeitwald wechseln. Darüberhinaus ist das simultane Tuning numerischer Parameter, etwa zur Vorverarbeitung, Modell-basierten

Optimierung gegenüber der Analysestrategie CoxBoost problemlos möglich. Dies alles spricht dafür, stets das die klassische Modell-basierte Optimierung als Analysestrategie zu wählen.

7 Zusammenfassung

Die Überlebenszeitanalyse hochdimensionaler Genexpressionsdaten stellt ein aktuelles und aktives Forschungsgebiet dar. Während stetig zahlreiche neue Vorschläge zur Vorverarbeitung, Variablenselektion, Modellierung und Enrichment der Genexpressionen mit zusätzlicher Information erarbeitet werden, mangelt es an aussagekräftigen Vergleichen der bestehenden Verfahren. Derzeit orientiert sich die Auswertung in der Praxis daher an wenigen kleinen, nicht repräsentativen Studien. Zwar ist das Methoden-Portfolio in diesen Studien oft recht umfangreich, jedoch erfolgt die Validierung auf nur wenigen Datensätzen, und dort teilweise nur auf einer einzigen Aufteilung in Trainingsmenge und Testmenge. Darüber hinaus findet die Hyperparameteroptimierung von Vorverarbeitungsmethoden und Modellen bisher kaum Berücksichtigung. Welche Methoden oder Kombination von Methoden mit welcher Parametrisierung verlässlich zu Modellen mit einer hohen prädiktiver Güte führen, ist somit vollkommen unklar. Das Fehlen guter Vergleichsstudien ist darin begründet, dass für die Durchführung mehrere Aspekte zusammenkommen müssen. Zunächst ist offensichtlich ein fundierter Überblick über die Methoden der Überlebenszeitanalyse erforderlich. Weiter müssen moderne Verfahren zur Hyperparameteroptimierung eingesetzt werden und eine parallele Ausführung auf Hochperformanz-Clustern ist aufgrund der hohen Ressourcenanforderungen der Überlebenszeitmodelle zudem unausweichlich. Dabei setzen alle drei aufgeführten Aspekte gute Programmierkenntnisse und sauberes Software-Design voraus. Deshalb wurde für diese Arbeit die Unterstützung für die Überlebenszeitanalyse in das Paket `mlr` eingepflegt, in dem Paket `mlrMBO` die Entwicklung eines modularen Frameworks für die Modell-basierte Optimierung vorangetrieben durch `BatchJobs` und `BatchExperiments` eine Abstraktion für parallel auszuführende Computerexperimente geschaffen. Denn ohne diese Komponenten wäre die Durchführung einer Benchmarkstudie des in dieser Arbeit vorgestellten Umfangs nicht möglich.

In dieser Arbeit wird durch Verwendung der Modell-basierten Optimierung ein Vorschlag zur automatischen Modellwahl und Modellkonfiguration präsentiert. In Kapitel 2 ist

zunächst ein Überblick über den aktuellen Stand der Forschung gegeben, wobei die eigene Arbeit bezüglich der Forschungsgebiete der Überlebenszeitanalyse und der Hyperparameteroptimierung eingeordnet wird. Kapitel 3 bietet eine kurze Einführung in die Überlebenszeitanalyse und stellt das wohl wichtigste Modell in diesem Feld vor, das Cox Proportional Hazard-Modell. Erweiterungen des Cox-Modells wie Boosting oder Regularisierung werden thematisiert, ebenso wie Überlebenszeitbäume und deren Ensemble, Überlebenszeitwälder. Filter zur Vorselektion von Variablen sowie Gütemaße für zensierte Beobachtungen werden ebenfalls in Kapitel 3 behandelt. Das folgende Kapitel 4 beschäftigt sich mit der Optimierung. Dazu wird das allgemeine Optimierungsproblem der Algorithmenkonfiguration zunächst formal definiert und anschließend auf das speziellere Problem der Hyperparameteroptimierung eingegangen. Mit der Modellbasierten Optimierung wird ein modernes Verfahren zur Konfiguration von Algorithmen des Maschinellen Lernens vorgestellt. Dabei wird ausführlich auf die besonderen Anforderungen von gemischt-skalierten Parameterräumen eingegangen. Die Konstruktion eines fusionierten Lernverfahrens aus beliebigen Basislernen zu einem Modellmultiplexer als Eingabe für das Optimierungsverfahren schlägt die Brücke zur Modellwahl. Im folgenden Kapitel 5 wird mit dem Map-Reduce-Paradigma eines der wichtigsten Konzepte der parallelen Programmierung vorgestellt. Nach einem kurzen Überblick über geeignete Ausführungsplattformen wird die Funktionsweise des `BatchJobs`-Pakets zur Parallelisierung auf Hochperformanz-Clustern dargestellt. Mit `BatchExperiments` wird eine Erweiterung von `BatchJobs` zur Abstraktion allgemeiner Computerexperimente präsentiert. Zudem werden Pakete vorgestellt, welche eine Abstraktion einer *Map*-Operation für verschiedene Backends zur parallelen Ausführung unterstützen. Das Kapitel 5 schließt mit Anmerkungen zur Reproduzierbarkeit von Computerexperimenten im Allgemeinen, und parallelen Computerexperimenten im Speziellen. Die Beschreibung und Auswertung des durchgeführten Vergleichs verschiedener Analysestrategien findet sich in Kapitel 6. Darin wird zunächst ein deskriptiver Überblick über die sieben ausgewerteten Lungenkrebsdatensätze gegeben und auch die Kriterien zur Datenakquise sowie die notwendige Vorverarbeitung berücksichtigt. Es folgt eine detailierte Beschreibung des Versuchsaufbaufs. Neben einem Überblick zum Resampling werden Methoden zur Validierung der automatischen Modellwahl mittels Modell-basierter Optimierung betrachtet. Die Optimierung erfolgt über den Raum aller Modelle und aller Filter sowie jeweils zugehöriger Hyperparameter. Die Random Search wird als alternatives Optimierungsverfahren zum Vergleich herangezogen. Darüber hinaus wird mit der aktuell üblichen Praxis verglichen, entweder stets ein bestimmtes automatisch getuntes Modell a-priori zu wählen oder alternativ eine kleine

Benchmarkstudie sich selbst optimierender Verfahren durchzuführen.

Die grafische Analyse der Konkordanzindizes, berechnet auf unabhängigen Testdaten, deckt weitestgehend keine Auswertungsstrategien als deutlich überlegen oder unterlegen auf. Lediglich die Lasso-Regression fällt hier negativ auf. Jedoch zeichnen sich einige Eigenschaften der Datensätze deutlich ab. Auf den beiden kleinsten der insgesamt sieben Datensätze greift überhaupt keine Auswertungsmethode. Ob dies an der Anzahl Patienten bzw. Ereignisse oder an der Qualität der Daten liegt, ist ungewiss. Auf fast allen Datensätzen wird zudem deutlich, dass die Aufteilung des Resamplings einen sehr großen Einfluss auf die resultierende Güte besitzt.

Weil die grafische Auswertung kein klares Bild zeichnet, wird auf die Betrachtung von gemittelten Rangstatistiken zurückgegriffen. Dabei stellt sich die in dieser Arbeit vorgestellte Modell-basierte Optimierung in der klassischen Variante als vielversprechendste Auswertungsstrategie heraus. Diese Variante setzt sich klar gegen eine alternierende Variante durch, welche in jedem zweiten Schritt eine forcierte Exploration durchführt. Die Random Search schneidet dabei im Vergleich noch schlechter ab. Eine einfache Alternative zur Modell-basierten Optimierung scheinen Boosting-Ansätze zu bieten, welche eine vergleichbare Modellgüte aufweisen, jedoch weniger flexibel sind und nicht ohne Weiteres mit einer parametrisierten Vorverarbeitungsmethode kombinierbar sind. Die Strategie, eine kleine Benchmark-Studie zur Wahl eines geeigneten Referenzmodells durchzuführen, stellt insgesamt die zweitschlechteste Auswertungsstrategie dar. Lediglich die a-priori Festlegung auf eine Lasso-Regression ist noch schlechter. Dies ist eine sehr wichtige Erkenntnis, da die Benchmark-Optimierung den de-facto Standard in der aktuellen anwendungsorientierten Auswertung darstellt. Auch die L_1-Regression wird häufig eingesetzt, wobei die Regularisierung dabei so eingestellt wird, dass eine zuvor bestimmte Anzahl an Kovariablen im Modell verbleibt. Dabei kann die aus dieser Art der Konfiguration resultierende Modellgüte als noch schlechter als in dieser Arbeit beobachtet angenommen werden. Somit können zwei in der Praxis sehr beliebte Ansätze als fraglich bezeichnet werden.

In der Zukunft sollten Experimente, wie sie in dieser Arbeit vorgestellt wurden, regelmäßig unter Berücksichtigung der neusten Erkenntnisse und Modellierungsoptionen durchgeführt und veröffentlicht werden. Durch die implementierte Funktionalität in `mlr`, `mlrMBO`, `BatchJobs` und `BatchExperiments` ist dafür der Grundstein gelegt worden. Natürlich wäre eine noch größere Datenbasis dabei wünschenswert. Insbesondere bleibt fraglich, inwiefern sich Patienten verschiedener Tumorsubtypen voneinander unterscheiden und

ob die verwendeten Modelle diese zuverlässig voreinander trennen können, um gegebenenfalls separate bzw. spezialisierte Prognosemodelle anzupassen. Bei Zufallsbäumen und Zufallswäldern ist die Möglichkeit einer klaren Trennung offensichtlich gegeben, bei den Modellen basierend auf dem CoxPH-Modell erscheint die Verschiebung um einen linearen Term zum Ausgleich großer systematischer Unterschiede als eher unzureichend anpassungsfähig. Hier drängt sich eher die Hinzunahme von Methoden der Subgruppenanalyse in das Benchmarking-Framework auf. Die Modellierung unter Berücksichtigung von Subgruppen bietet dabei gleichzeitig einen Ansatz zur Suche nach einem Modell, welches für eine ganze Domäne sinnvolle Prognosen liefern kann. Denn nur wenn Unterschiede zwischen Gruppen, seien es Tumortypen, das Geschlecht oder ganze Kohorten, adäquat in der Modellierung Berücksichtigung finden, kann hier Generalisierbarkeit erwartet werden. Ebenso ist natürlich eine Aufnahme weiterer Vorverarbeitungsmethoden in die Optimierung sinnvoll. Außerdem steht eine Validierung mit einem anderen Maß als dem Konkordanzindex noch aus. Als Gegenentwurf zur ausschließlichen Betrachtung der Rangfolge vorhergesagter Überlebenszeiten basiert der Brier-Score auf einer Summe quadrierter Abweichungen zwischen beobachteter und vorhergesagter Überlebenszeiten. Die Betrachtung eines zweiten Maßes ist in der Überlebenszeitanalyse wichtig, da verschiedene Maße gerade auf schwierig zu modellierenden Datensätzen zu widersprüchlichen Aussagen führen können. Allerdings gestaltet sich die Berechnung des Brier-Scores als technisch schwierig, eine Unterstützung durch `mlr` steht hier noch aus.

Eine regelmäßige Überprüfung des aktuellen Stands alleine würde die hochdimensionale Überlebenszeitanalyse schon deutlich voran treiben. Ein weiterer großer Schritt in Richtung reproduzierbarer Wissenschaft kann etwa durch OpenML (Rijn u. a., 2013) erfolgen. OpenML erlaubt die standardisierte Speicherung und Bereitstellung von Daten, zugehörigen Resampling-Aufteilungen, Algorithmen und erreichten Gütemaßen. Damit erlaubt OpenML ein kollaboratives Benchmarking. Die favorisierte Auswertungsstrategien lässt sich hier einfach publizieren und systematisch mit alternativen Vorgehen vergleichen. Eine Anbindung an `R` findet sich im Paket `OpenML` (Bischl u. a., 2015), wobei die Unterstützung für Überlebenszeitdaten und Überlebenszeitmodelle bereits serverseitig vorliegt, jedoch noch nach ausführlicheren Tests verlangt.

Aber auch die Modell-basierte Optimierung bietet noch viele Ansatzpunkte mit Raum für Verbesserungen. Einerseits ist die Unsicherheitsschätzung des Surrogatmodells nicht ganz unproblematisch. Hier gilt es, entweder alternative Regressionsmodelle mit geeigneterer Varianzschätzung für gemischt-skalierte Hyperparameterräume zu finden, oder die Varianzschätzung durch ein Maß, welches beispielsweise auf der Gower-Distanz (Gower, 1971)

basiert, zu substituieren. Auch Anpassungen an das sehr heuristischen Infill-Kriterium LCB sind denkbar. Eine sehr einfache, aber vielversprechende erste Verbesserung könnte hier eine adaptive Wahl des Balancierungsterm λ_{LCB} sein. Anstatt ihn stets aus einer konstant parametrisierten Exponentialverteilung zu ziehen, sollte das Budget berücksichtigt werden, so dass bei ausreichend Budget mehr Wert auf Exploration und bei wenig Budget adaptiv mehr Wert auf Exploitation gelegt wird.

Darüberhinaus sollten Ressourcenkosten explizit in der Optimierung berücksichtigt werden. Wird nicht mehr an dem Konzept eines festen Budgets festgehalten, sondern stattdessen eine Zeitschranke betrachtet, bis zu der eine möglichst gute Konfiguration gefunden werden muss, so ist eine Fokusierung auf günstige Regionen des Hyperparameterraums sinnvoll. Solch eine Präferenz kann beispielsweise durch eine Adaption des Infill-Kriterium explizit kodiert werden. Ebenso kann mit Unterstützung eines Ressourcenmodells die parallele Ausführung effektiver geplant und auf die verfügbaren Ressourcen des Ausführungssystems abgestimmt werden. Arbeiten an den Optimierungsverfahren müssen jedoch auf Problemen untersucht werden, welche durch geringere Anforderungen an Laufzeit und Speicherverbrauch eine systematische Untersuchung der Neuerungen erlauben.

Literatur

Aalen, Odd (1978). „Nonparametric inference for a family of counting processes“. In: *The Annals of Statistics* 6.4, S. 701–726. DOI: 10.1214/aos/1176344247.

Amazon.com (2014). *Amazon Elasic Cloud Computing (EC2)*. URL: https://aws.amazon.com/ec2/.

Anderson, Edgar (1935). „The irises of the Gaspe Peninsula“. In: *Bulletin of the American Iris society* 59, S. 2–5.

Apache Foundation (2014). *Apache Hadoop*. URL: http://hadoop.apache.org/.

Benner, Axel, Manuela Zucknick, Thomas Hielscher, Carina Ittrich und Ulrich Mansmann (2010). „High-Dimensional Cox Models: The Choice of Penalty as Part of the Model Building Process“. In: *Biometrical Journal* 52.1, S. 50–69. DOI: 10.1002/bimj.200900064.

Bergstra, James und Yoshua Bengio (2012). „Random search for hyper-parameter optimization“. In: *The Journal of Machine Learning Research* 13.1, S. 281–305. URL: http://dl.acm.org/citation.cfm?id=2188395.

Binder, Harald und Martin Schumacher (2008a). „Adapting prediction error estimates for biased complexity selection in high-dimensional bootstrap samples“. In: *Statistical Applications in Genetics and Molecular Biology* 7.1. DOI: 10.2202/1544-6115.1346.

Binder, Harald und Martin Schumacher (2008b). „Allowing for mandatory covariates in boosting estimation of sparse high-dimensional survival models“. In: *BMC Bioinformatics* 9.1, S. 14. DOI: 10.1186/1471-2105-9-14.

Birattari, Mauro, Thomas Stützle, Luis Paquete und Klaus Varrentrapp (2002). „A Racing Algorithm for Configuring Metaheuristics.“ In: *GECCO*. Bd. 2, S. 11–18. DOI: 10.1.1.70.2018.

Bischl, Bernd und Michel Lang (2014). *parallelMap: Unified interface to some popular parallelization back-ends for interactive usage and package development*. URL: http://cran.r-project.org/web/packages/parallelMap/index.html.

Bischl, Bernd, Olaf Mersmann und Heike Trautmann (2010). „Resampling methods in model validation“. In: *Workshop on Experimental Methods for the Assessment of Computational Systems (WEMACS 2010)*, S. 14. DOI: 10.1162/EVCO_a_00069.

Bischl, Bernd, Michel Lang, Olaf Mersmann, Joerg Rahnenfuehrer und Claus Weihs (2012a). *Computing on high performance clusters with R: Packages BatchJobs and BatchExperiments*. SFB876 Technical Report 1. TU Dortmund University. URL: http://sfb876.tu-dortmund.de/PublicPublicationFiles/bischl_etal_2012a.pdf.

Bischl, Bernd, Olaf Mersmann, Heike Trautmann und Claus Weihs (2012b). „Resampling methods for meta-model validation with recommendations for evolutionary computation“. In: *Evolutionary computation* 20.2, S. 249–275. DOI: 10.1162/EVCO_a_00069.

Bischl, Bernd, Michel Lang, Jakob Richter, Jakob Bossek, Leonard Judt, Tobias Kühn und Lars Kotthoff (2014a). *mlr: Machine Learning in R*. URL: https://github.com/berndbischl/mlr.

Bischl, Bernd, Jakob Bossek, Daniel Horn und Michel Lang (2014b). *Model-Based Optimization for mlr*. URL: https://github.com/berndbischl/mlrMBO.

Bischl, Bernd, Simon Wessing, Nadja Bauer, Klaus Friedrichs und Claus Weihs (2014c). „MOI-MBO: Multiobjective Infill for Parallel Model-Based Optimization“. In: *Learning and Intelligent Optimization Conference*. Florida. DOI: 10.1007/978-3-319-09584-4_17.

Bischl, Bernd, Michel Lang, Jakob Bossek und Daniel Horn (2014d). *ParamHelpers: Helpers for parameters in black-box optimization, tuning and machine learning*. URL: http://cran.r-project.org/web/packages/ParamHelpers.

Bischl, Bernd, Luis Torgo, Dominik Kirchhoff und Michel Lang (2015). *OpenML*. URL: https://github.com/openml/r.

Bøvelstad, Hege M., Ståle Nygård, Hege L. Størvold, Magne Aldrin, Ørnulf Borgan, Arnoldo Frigessi und Ole Christian Lingjærde (2007). „Predicting survival from microarray data - a comparative study“. In: *Bioinformatics* 23.16, S. 2080–2087. DOI: 10.1093/bioinformatics/btm305.

Breiman, Leo (1996). „Bagging predictors“. In: *Machine learning* 24.2, S. 123–140. DOI: 10.1007/BF00058655.

Breiman, Leo (2001). „Random forests“. In: *Machine learning* 45.1, S. 5–32. DOI: 10.1023/A:1010933404324.

Breiman, Leo, Jerome H. Friedman, Richard A. Olshen und Charles J. Stone (1984). *Classification and Regression Trees*. Chapman & Hall, NY. ISBN: 978-0412048418.

Brier, Glenn W. (1950). „Verification of forecasts expressed in terms of probability“. In: *Monthly weather review* 78.1, S. 1–3. DOI: 10.1175/1520-0493(1950)078<0001:VOFEIT>2.0.CO;2.

Bühlmann, Peter und Torsten Hothorn (2007). „Boosting algorithms: Regularization, prediction and model fitting“. In: *Statistical Science*, S. 477–505. DOI: 10.1214/07-STS242.

Bühlmann, Peter und Bin Yu (2003). „Boosting with the L2 loss: regression and classification“. In: *Journal of the American Statistical Association* 98.462, S. 324–339. DOI: 10.1198/016214503000125.

Cox, David R (1972). „Regression models and life-tables“. In: *Journal of the Royal Statistical Society. Series B (Methodological)*, S. 187–220. URL: http://www.jstor.org/stable/2985181.

De Bin, Ricardo, Willi Sauerbrei und Anne-Laure Boulesteix (2014). *Investigating the prediction ability of survival models based on both clinical and omics data: two case studies.* Techn. Ber. 153. München: Department of Statistics, LMU München. URL: http://epub.ub.uni-muenchen.de/18386/7/techn Rep_153.pdf.

Ding, Chris und Hanchuan Peng (2005). „Minimum redundancy feature selection from microarray gene expression data“. In: *Journal of bioinformatics and computational biology* 3.02, S. 185–205. DOI: 10.1142/S0219720005001004.

Dowle, Matt, Steve Lianoglou und Arun Srinivasan (2014). *data.table: Extension of data.frame.* URL: http://CRAN.R-project.org/package=data.table.

Eddelbuettel, Dirk (2014). *CRAN Task View: High-Performance and Parallel Computing with R.* URL: http://cran.r-project.org/web/views/HighPerformanceComputing.html.

Eddelbuettel, Dirk und Conrad Sanderson (2014). „RcppArmadillo: Accelerating R with high-performance C++ linear algebra“. In: *Computational Statistics & Data Analysis* 71, S. 1054–1063. DOI: 10.1016/j.csda.2013.02.005.

Edgar, Ron, Michael Domrachev und Alex E Lash (2002). „Gene Expression Omnibus: NCBI gene expression and hybridization array data repository“. In: *Nucleic acids research* 30.1, S. 207–210. DOI: 10.1093/nar/30.1.207.

Efron, Bradley (1977). „The Efficiency of Cox's Likelihood Function for Censored Data“. In: *Journal of the American Statistical Association* 72.359, S. 557–565. DOI: 10.2307/2286217.

Freund, Yoav und Robert E. Schapire (1995). „A desicion-theoretic generalization of on-line learning and an application to boosting“. In: *Computational learning theory*, S. 23–37. DOI: 10.1007/3-540-59119-2_166.

Friedman, Jerome H. (2001). „Greedy function approximation: a gradient boosting machine“. In: *Annals of Statistics*, S. 1189–1232. DOI: 10.1214/aos/1013203451.

Friedman, Jerome H. (2002). „Stochastic gradient boosting“. In: *Computational Statistics & Data Analysis* 38.4, S. 367–378. DOI: 10.1016/S0167-9473(01)00065-2.

Friedman, Jerome H., Trevor Hastie und Robert Tibshirani (2010). „Regularization paths for generalized linear models via coordinate descent“. In: *Journal of statistical software* 33.1. URL: http://www.jstatsoft.org/v33/i01.

Gentleman, Robert C., Vincent J. Carey, Douglas M. Bates u. a. (2004). „Bioconductor: open software development for computational biology and bioinformatics“. In: *Genome Biology* 5.10, R80. DOI: 10.1186/gb-2004-5-10-r80.

Gentzsch, Wolfgang (2001). „Sun grid engine: Towards creating a compute power grid“. In: *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*. IEEE, S. 35–36. DOI: 10.1109/CCGRID.2001.923173.

Goeman, Jelle J. (2010). „L1 penalized estimation in the cox proportional hazards model“. In: *Biometrical Journal* 52.1, S. 70–84. DOI: 10.1002/bimj.200900028.

Gower, John C. (1971). „A General Coefficient of Similarity and Some of Its Properties“. In: *Biometrics* 27.4, S. 857–871. DOI: 10.2307/2528823.

Gramacy, Robert B. und Herbert KH Lee (2008). „Bayesian treed Gaussian process models with an application to computer modeling“. In: *Journal of the American Statistical Association* 103.483. DOI: 10.1198/016214508000000689.

Gropp, William, Ewing Lusk und Anthony Skjellum (1999). *Using MPI: portable parallel programming with the message-passing interface*. 2. Aufl. Bd. 1. MIT press. ISBN: 0-262-57132-3.

Heagerty, Patrick J. und Yingye Zheng (2005). „Survival Model Predictive Accuracy and ROC Curves“. In: *Biometrics* 61.1, S. 92–105. DOI: 10.1111/j.0006-341X.2005.030814.x.

Henderson, Robert L. (1995). „Job scheduling under the portable batch system“. In: *Job scheduling strategies for parallel processing*. Springer, S. 279–294. DOI: 10.1007/3-540-60153-8_34.

Hoerl, Arthur E. und Robert W. Kennard (1970). „Ridge regression: Biased estimation for nonorthogonal problems“. In: *Technometrics* 12.1, S. 55–67. DOI: 10.1080/00401706.1970.10488634.

Horn, Daniel, Tobias Wagner, Dirk Biermann, Claus Weihs und Bernd Bischl (2015). „Model-Based Multi-Objective Optimization: Taxonomy, Multi-Point Proposal, Toolbox and Benchmark (accepted)“. In: *Evolutionary Multi-Criterion Optimization (EMO)*. Lecture Notes in Computer Science.

Hothorn, Torsten (2014). *TH.data: TH's Data Archive*. URL: http://CRAN.R-project.org/package=TH.data.

Hothorn, Torsten, Berthold Lausen, Axel Benner und Martin Radespiel-Tröger (2004). „Bagging survival trees". In: *Statistics in Medicine* 23.1, S. 77–91. DOI: 10.1002/sim.1593.

Hothorn, Torsten, Peter Bühlmann, Sandrine Dudoit, Annette Molinaro und Mark J. Van Der Laan (2006). „Survival ensembles". In: *Biostatistics* 7.3, S. 355–373. DOI: 10.1093/biostatistics/kxj011.

Hothorn, Torsten, Peter Bühlmann, Thomas Kneib, Matthias Schmid und Benjamin Hofner (2010). „Model-based boosting 2.0". In: *The Journal of Machine Learning Research* 99, S. 2109–2113. URL: http://dl.acm.org/citation.cfm?id=1859922.

Hudis, Clifford A., William E. Barlow, Joseph P. Costantino, Robert J. Gray, Kathleen I. Pritchard, Judith-Anne W. Chapman, Joseph A. Sparano, Sally Hunsberger, Rebecca A. Enos, Richard D. Gelber und Jo Anne Zujewski (2007). „Proposal for Standardized Definitions for Efficacy End Points in Adjuvant Breast Cancer Trials: The STEEP System". In: *Journal of Clinical Oncology* 25.15, S. 2127–2132. DOI: 10.1200/JCO.2006.10.3523.

Hutter, Frank, Holger H. Hoos und Kevin Leyton-Brown (2011). „Sequential Model-Based Optimization for General Algorithm Configuration". In: *Learning and Intelligent Optimization*. Hrsg. von Carlos A. Coello Coello. Lecture Notes in Computer Science 6683. Springer Berlin Heidelberg, S. 507–523. ISBN: 978-3-642-25565-6.

Irizarry, Rafael A., Bridget Hobbs, Francois Collin, Yasmin D. Beazer-Barclay, Kristen J. Antonellis, Uwe Scherf und Terence P. Speed (2003). „Exploration, normalization, and summaries of high density oligonucleotide array probe level data". In: *Biostatistics* 4.2, S. 249–264. DOI: 10.1093/biostatistics/4.2.249.

Ishwaran, Hemant, Udaya B. Kogalur, Eugene H. Blackstone und Michael S. Lauer (2008). „Random survival forests". In: *The Annals of Applied Statistics* 2.3, S. 841–860. DOI: 10.1214/08-AOAS169.

Järvinen, Anna-Kaarina, Sampsa Hautaniemi, Henrik Edgren, Petri Auvinen, Janna Saarela, Olli-P. Kallioniemi und Outi Monni (2004). „Are data from different gene expression microarray platforms comparable?" In: *Genomics* 83.6, S. 1164–1168. DOI: 10.1016/j.ygeno.2004.01.004.

Jones, Donald R., Matthias Schonlau und William J. Welch (1998). „Efficient global optimization of expensive black-box functions". In: *Journal of Global optimization* 13.4, S. 455–492. DOI: 10.1023/A:1008306431147.

Kammers, Kai, Michel Lang, Jan G. Hengstler, Marcus Schmidt und Jörg Rahnenführer (2011). „Survival models with preclustered gene groups as covariates“. In: *BMC Bioinformatics* 12.1, S. 478. DOI: 10.1186/1471-2105-12-478.

Kaplan, Edward L. und Paul Meier (1958). „Nonparametric estimation from incomplete observations“. In: *Journal of the American statistical association* 53.282, S. 457–481. URL: http://www.jstor.org/stable/2281868.

Kearns, Michael und Leslie Valiant (1994). „Cryptographic Limitations on Learning Boolean Formulae and Finite Automata“. In: *J. ACM* 41.1, S. 67–95. DOI: 10.1145/174644.174647.

Klein, John P. und Melvin L. Moeschberger (2003). *Survival analysis: techniques for censored and truncated data*. 2nd ed. Statistics for biology and health. New York: Springer. ISBN: 038795399X.

Knaus, Jochen (2013). *snowfall: Easier cluster computing (based on snow)*. URL: http://CRAN.R-project.org/package=snowfall.

Kratz, Johannes R., Jianxing He, Stephen K. Van Den Eeden, Zhi-Hua Zhu, Wen Gao, Patrick T. Pham, Michael S. Mulvihill, Fatemeh Ziaei, Huanrong Zhang und Bo Su (2012). „A practical molecular assay to predict survival in resected non-squamous, non-small-cell lung cancer: development and international validation studies“. In: *The Lancet* 379.9818, S. 823–832. DOI: 10.1016/S0140-6736(11)61941-7.

Krige, D. G. (1951). „A statistical approach to some mine valuation and allied problems on the Witwatersrand“. Diss. University of the Witwatersrand.

Laan, Mark J. Van der und James M. Robins (2003). *Unified methods for censored longitudinal data and causality*. Springer. ISBN: 978-1-4419-3055-2.

Lang, Michel (2014). *fmrmr: Fast feature selection using mRMR*. URL: https://github.com/mllg/fmrmr.

Lang, Michel, Helena Kotthaus, Peter Marwedel, Claus Weihs, Jörg Rahnenführer und Bernd Bischl (2015). „Automatic model selection for high-dimensional survival analysis“. In: *Journal of Statistical Computation and Simulation* 85.1, S. 62–76. DOI: 10.1080/00949655.2014.929131.

LeBlanc, Michael und John Crowley (1993). „Survival Trees by Goodness of Split“. In: *Journal of the American Statistical Association* 88.422, S. 457–467. DOI: 10.2307/2290325.

Lee, Sangkyun, Jörg Rahnenführer, Michel Lang, Katleen De Preter, Pieter Mestdagh, Jan Koster, Rogier Versteeg, Raymond L. Stallings, Luigi Varesio, Shahab Asgharzadeh, Johannes H. Schulte, Kathrin Fielitz, Melanie Schwermer, Katharina Morik und

Alexander Schramm (2014). „Robust selection of cancer survival signatures from high-throughput genomic data using two-fold subsampling“. In: *PloS One* 9.10, e108818. DOI: 10.1371/journal.pone.0108818.

Liaw, Andy und Matthew Wiener (2002). „Classification and Regression by randomForest“. In: *R News* 3.2, S. 18–22. ISSN: 2073-4859. URL: http://cran.r-project.org/doc/Rnews/Rnews_2003-2.pdf.

Lindstrom, Mary J. und Douglas M. Bates (1988). „Newton-Raphson and EM Algorithms for Linear Mixed-Effects Models for Repeated-Measures Data“. In: *Journal of the American Statistical Association* 83.404, S. 1014–1022. DOI: 10.1080/01621459.1988.10478693.

López-Ibánez, Manuel, Jérémie Dubois-Lacoste, Thomas Stützle und Mauro Birattari (2011). *The irace package, iterated race for automatic algorithm configuration*. Technical Report TR/IRIDIA/2011-004. URL: http://iridia.ulb.ac.be/IridiaTrSeries/IridiaTr2011-004.pdf.

Mantel, Nathan (1963). „Chi-square tests with one degree of freedom; extensions of the Mantel-Haenszel procedure“. In: *Journal of the American Statistical Association* 58.303, S. 690–700. DOI: 10.1080/01621459.1963.10500879.

Matsumoto, Makoto und Takuji Nishimura (1998). „Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator“. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1, S. 3–30. DOI: 10.1145/272991.272995.

McCall, Matthew N., Benjamin M. Bolstad und Rafael A. Irizarry (2010). „Frozen robust multiarray analysis (fRMA)“. In: *Biostatistics* 11.2, S. 242–253. DOI: 10.1093/biostatistics/kxp059.

Molinaro, Annette M., Richard Simon und Ruth M. Pfeiffer (2005). „Prediction error estimation: a comparison of resampling methods“. In: *Bioinformatics* 21.15, S. 3301–3307. DOI: 10.1093/bioinformatics/bti499.

Morgan, Martin, Ryan Thompson und Michel Lang (2014). *BiocParallel: Bioconductor facilities for parallel evaluation*. URL: http://www.bioconductor.org/packages/devel/bioc/html/BiocParallel.html.

Nelson, Wayne (1972). „Theory and Applications of Hazard Plotting for Censored Failure Data“. In: *Technometrics* 14.4, S. 945–966. DOI: 10.1080/00401706.1972.10488991.

Peng, Hanchuan, Fuhui Long und Chris Ding (2005). „Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy“. In:

Pattern Analysis and Machine Intelligence, IEEE Transactions on 27.8, S. 1226–1238. DOI: 10.1109/TPAMI.2005.159.

R Core Team (2014). *R: A language and environment for statistical computing*. URL: http://www.R-project.org.

Rasmussen, Carl Edward (2004). „Gaussian Processes in Machine Learning“. In: *Advanced Lectures on Machine Learning*. Hrsg. von Olivier Bousquet, Ulrike von Luxburg und Gunnar Rätsch. Lecture Notes in Computer Science 3176. Springer Berlin Heidelberg, S. 63–71. ISBN: 978-3-540-23122-6.

Revolution Analytics und Steve Weston (2014). *foreach: Foreach looping construct for R*. URL: http://CRAN.R-project.org/package=foreach.

Rijn, Jan N. van, Bernd Bischl, Luis Torgo, Bo Gao, Venkatesh Umaashankar, Simon Fischer, Patrick Winter, Bernd Wiswedel, Michael R. Berthold und Joaquin Vanschoren (2013). „OpenML: A Collaborative Science Platform“. In: *Machine Learning and Knowledge Discovery in Databases*. Hrsg. von Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen und Filip Železný. Lecture Notes in Computer Science 8190. Springer Berlin Heidelberg, S. 645–649. ISBN: 978-3-642-40993-6.

Sanderson, Conrad (2010). *Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments*. NICTA Technical Report 2. URL: http://arma.sourceforge.net/armadillo_nicta_2010.pdf.

Schapire, Robert E. (1990). „The strength of weak learnability“. In: *Machine Learning* 5.2, S. 197–227. DOI: 10.1007/BF00116037.

Schena, Mark, Dari Shalon, Ronald W. Davis und Patrick O. Brown (1995). „Quantitative monitoring of gene expression patterns with a complementary DNA microarray“. In: *Science* 270.5235, S. 467–470. DOI: 10.1126/science.270.5235.467.

Schumacher, Martin, Harald Binder und Thomas Gerds (2007). „Assessment of survival prediction models based on microarray data“. In: *Bioinformatics* 23.14, S. 1768–1774. DOI: 10.1093/bioinformatics/btm232.

Sexton, Joseph und Petter Laake (2009). „Standard errors for bagged and random forest estimators“. In: *Computational Statistics & Data Analysis*. Computational Statistics within Clinical Research 53.3, S. 801–811. DOI: 10.1016/j.csda.2008.08.007.

Shedden, Kerby, Jeremy M G Taylor, Steven A Enkemann u. a. (2008). „Gene expression–based survival prediction in lung adenocarcinoma: a multi-site, blinded validation study“. In: *Nature Medicine* 14.8, S. 822–827. DOI: 10.1038/nm.1790.

Simon, Noah, Jerome Friedman, Trevor Hastie und Robert Tibshirani (2011). „Regularization paths for Cox's proportional hazards model via coordinate descent". In: *Journal of statistical software* 39.5, S. 1–13. URL: http://www.jstatsoft.org/v39/i05/.

Snoek, Jasper, Hugo Larochelle und Ryan P. Adams (2012). „Practical Bayesian Optimization of Machine Learning Algorithms". In: *NIPS workshop on Bayesian optimization, sequential experimental design, and bandits*, S. 2960–2968. URL: https://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

Solis, Francisco J. und Roger J.-B. Wets (1981). „Minimization by random search techniques". In: *Mathematics of operations research* 6.1, S. 19–30. DOI: 10.1287/moor.6.1.19.

Stein, Michael (1987). „Large Sample Properties of Simulations Using Latin Hypercube Sampling". In: *Technometrics* 29.2, S. 143–151. DOI: 10.1080/00401706.1987.10488205.

Therneau, Terry M. und Patricia M. Grambsch (2000). *Modeling survival data: extending the Cox model*. Statistics for biology and health. New York: Springer. ISBN: 0387987843.

Thornton, Chris, Frank Hutter, Holger H. Hoos und Kevin Leyton-Brown (2012). „Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms". In: *arXiv preprint arXiv:1208.3719*. URL: http://arxiv.org/abs/1208.3719.

Thornton, Chris, Frank Hutter, Holger H. Hoos und Kevin Leyton-Brown (2013). „Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, S. 847–855. DOI: 10.1145/2487575.2487629.

Tibshirani, Robert (1996). „Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1, S. 267–288. URL: http://www.jstor.org/stable/2346178.

Tibshirani, Robert (1997). „The lasso method for variable selection in the Cox model". In: *Statistics in medicine* 16.4, S. 385–395. DOI: 10.1002/(SICI)1097-0258(19970228)16:4<385::AID-SIM380>3.0.CO;2-3.

Tibshirani, Robert und Brad Efron (2002). „Pre-validation and inference in microarrays". In: *Statistical Applications in Genetics and Molecular Biology* 1.1. DOI: 10.2202/1544-6115.1000.

Tutz, Gerhard und Harald Binder (2006). „Generalized Additive Modeling with Implicit Variable Selection by Likelihood-Based Boosting". In: *Biometrics* 62.4, S. 961–971. DOI: 10.1111/j.1541-0420.2006.00578.x.

Van Heesch, Dimitri (2004). *Doxygen*. URL: http://www.stack.nl/~dimitri/doxygen/.

Van Wieringen, Wessel N., David Kun, Regina Hampel und Anne-Laure Boulesteix (2009). „Survival prediction using gene expression data: a review and comparison“. In: *Computational statistics & data analysis* 53.5, S. 1590–1603. DOI: doi:10.1016/j.csda.2008.05.021.

Verweij, Pierre JM und Hans C. Van Houwelingen (1993). „Cross-validation in survival analysis“. In: *Statistics in medicine* 12.24, S. 2305–2314. DOI: 10.1002/sim.4780122407.

Verweij, Pierre JM und Hans C. Van Houwelingen (1994). „Penalized likelihood in cox regression“. In: *Statistics in Medicine* 13.23-24, S. 2427–2436. DOI: 10.1002/sim.4780132307.

Wager, Stefan, Trevor Hastie und Bradley Efron (2014). „Confidence Intervals for Random Forests: The Jackknife and the Infinitesimal Jackknife“. In: *J. Mach. Learn. Res.* 15.1, S. 1625–1651. URL: http://jmlr.org/papers/v15/wager14a.html.

Wickham, Hadley, Peter Danenberg und Manuel Eugster (2014). *roxygen2: In-source documentation for R*. URL: http://CRAN.R-project.org/package=roxygen2.

Witten, Daniela M. und Robert Tibshirani (2009). „Survival analysis with high-dimensional covariates“. In: *Statistical Methods in Medical Research*. DOI: 10.1177/0962280209105024.

Yoo, Andy B., Morris A. Jette und Mark Grondona (2003). „SLURM: Simple linux utility for resource management“. In: *Job Scheduling Strategies for Parallel Processing*. Springer, S. 44–60. DOI: 10.1007/10968987_3.

Zhou, Songnian (1992). „LSF: Load Sharing in Large Heterogeneous Distributed Systems“. In: *International Workshop on Cluster Computing*. URL: ftp://www.learning.cs.toronto.edu/cs/ftp/public_html/public_html/dist/white/white-technical-reports/257/lsf.ps.Z.

Zou, Hui und Trevor Hastie (2005). „Regularization and variable selection via the elastic net“. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2, S. 301–320. DOI: 10.1111/j.1467-9868.2005.00503.x.

A Tabellen

Software	Quelle	Version bzw. Checkout
R	CRAN	v3.1.0
BatchJobs	GitHub	co:417534e0dd76826de073dd7dac7cb447d8e3bdd4
BatchExperiments	GitHub	co:a048af55a3a298143276d53220ccca84f3f35480
BBmisc	GitHub	co:72e60d59958de012e03ca73d438d5ceaf553af39
checkmate	CRAN	v1.5.1
CoxBoost	CRAN	v1.4
data.table	CRAN	v1.9.4
fmrmr	GitHub	co:ea3e5805b220372457ad2ac422aa09a1d3612c7f
glmnet	CRAN	v1.9-8
Hmisc	CRAN	v3.14-5
mboost	CRAN	v2.4-0
mlr	GitHub	co:72ed45949771859b565867f5a3b80bb41b9b51eb
mlrMBO	GitHub	co:57826a9fb9480054b555c55576bf549d38090e04
parallelMap	GitHub	co:cbc0a626c2f9cf75c6c297bd344e986c749bc299
ParamHelpers	GitHub	co:8a0363a5488d16a715fd20547606828345966940
penalized	CRAN	v0.9-42
randomForest	CRAN	v4.6-10
randomForestSRC	CRAN	v1.5.5
survival	CRAN	v2.37-7

Tabelle A.1: R-Version sowie Versionen für die Auswertung direkt verwendeter Pakete. Stabile Versionen abrufbar über CRAN (http://cran.r-project.org), Entwicklerversionen über GitHub. Die jeweiligen CRAN-Seiten enthalten Hyperlinks zu den entsprechenden GitHub-Seiten für die Entwicklerversion. Lediglich fmrmr ist nicht über CRAN verfügbar, sondern nur unter https://github.com/mllg/fmrmr zu finden.

Variante	Fold	C-Index	Hyperparameter
MBOK	1	0.481	Filter: **coxscore** (perc=0.7587) Modell: **CoxBoost** (stepno=387, penalty=968.2, stepsize.factor=1.364, sf.scheme=sigmoid)
MBOA	1	0.567	Filter: **fmrmr** (perc=0.5301) Modell: **CoxBoost** (stepno=482, penalty=340.5, stepsize.factor=1.211, sf.scheme=sigmoid)
RS	1	0.394	Filter: **coxscore** (perc=0.7459) Modell: **CoxBoost** (stepno=329, penalty=484.7, stepsize.factor=1.784, sf.scheme=sigmoid)
MBOK	2	0.494	Filter: **coxscore** (perc=0.003687) Modell: **mboost** (family=Weibull, mstop=223, nu=0.9549)
MBOA	2	0.483	Filter: **clinical** Modell: **rfsrc** (ntree=6, mtry.ratio=0.4551, nodesize=2, splitrule=logrankscore)
RS	2	0.552	Filter: **kratz.clinical** Modell: **mboost** (family=Weibull, mstop=265, nu=0.2285)
MBOK	3	0.560	Filter: **fmrmr** (perc=0.1597) Modell: **mboost** (family=CoxPH, mstop=226, nu=0.02681)
MBOA	3	0.353	Filter: **var** (perc=0.39) Modell: **CoxBoost** (stepno=45, penalty=455.1, stepsize.factor=0.4994, sf.scheme=sigmoid)
RS	3	0.431	Filter: **coxscore** (perc=0.8683) Modell: **CoxBoost** (stepno=225, penalty=377.7, stepsize.factor=1.898, sf.scheme=sigmoid)
MBOK	4	0.533	Filter: **var** (perc=0.02348) Modell: **CoxBoost** (stepno=50, penalty=976.4, stepsize.factor=1.888, sf.scheme=sigmoid)

MBOA	4	0.489	Filter: **kratz.clinical** Modell: **rfsrc** (ntree=144, mtry.ratio=0.7544, nodesize=1, splitrule=logrankscore)
RS	4	0.256	Filter: **cindex.train** (perc=0.6976) Modell: **mboost** (family=Weibull, mstop=487, nu=0.2796)
MBOK	5	0.567	Filter: **fmrmr** (perc=0.4216) Modell: **CoxBoost** (stepno=236, penalty=114, stepsize.factor=1.669, sf.scheme=sigmoid)
MBOA	5	0.442	Filter: **fmrmr** (perc=0.03522) Modell: **mboost** (family=CoxPH, mstop=244, nu=0.2305)
RS	5	0.442	Filter: **cindex.test** (perc=0.6584) Modell: **mboost** (family=CoxPH, mstop=68, nu=0.9344)

Tabelle A.2: Konfigurationen für Datensatz „GSE14814“ nach B = 3000 MBO-Iterationen. Konkordanzindex ausgewertet auf unabhängigen Testdaten der äußeren Kreuzvalidierung.

Variante	Fold	C-Index	Hyperparameter
MBOK	1	0.393	Filter: **cindex.test** (perc=0.9142) Modell: **CoxBoost** (stepno=206, penalty=973.3, stepsize.factor=1.251, sf.scheme=sigmoid)
MBOA	1	0.380	Filter: **fmrmr** (perc=0.1118) Modell: **CoxBoost** (stepno=443, penalty=165.2, stepsize.factor=0.8461, sf.scheme=sigmoid)
RS	1	0.372	Filter: **fmrmr** (perc=0.4364) Modell: **CoxBoost** (stepno=187, penalty=719.3, stepsize.factor=0.08224, sf.scheme=sigmoid)
MBOK	2	0.476	Filter: **fmrmr** (perc=0.01269) Modell: **mboost** (family=Weibull, mstop=241, nu=0.6594)

MBOA	2	0.582	Filter: **var** (perc=0.03284) Modell: **CoxBoost** (stepno=324, penalty=52.1, stepsize.factor=1.662, sf.scheme=sigmoid)
RS	2	0.566	Filter: **cindex.test** (perc=0.962) Modell: **mboost** (family=Weibull, mstop=182, nu=0.6866)
MBOK	3	0.509	Filter: **clinical** Modell: **mboost** (family=Weibull, mstop=481, nu=0.9896)
MBOA	3	0.479	Filter: **cindex.test** (perc=0.1375) Modell: **CoxBoost** (stepno=5, penalty=473.8, stepsize.factor=0.9461, sf.scheme=sigmoid)
RS	3	0.385	Filter: **cindex.train** (perc=0.5006) Modell: **CoxBoost** (stepno=266, penalty=855.1, stepsize.factor=0.1622, sf.scheme=sigmoid)
MBOK	4	0.568	Filter: **fmrmr** (perc=0.09008) Modell: **mboost** (family=Weibull, mstop=423, nu=0.06125)
MBOA	4	0.545	Filter: **coxscore** (perc=0.788) Modell: **mboost** (family=Weibull, mstop=296, nu=0.6813)
RS	4	0.619	Filter: **cindex.test** (perc=0.7423) Modell: **CoxBoost** (stepno=399, penalty=893.7, stepsize.factor=0.5351, sf.scheme=sigmoid)
MBOK	5	0.519	Filter: **fmrmr** (perc=0.7177) Modell: **mboost** (family=Weibull, mstop=98, nu=0.7422)
MBOA	5	0.519	Filter: **cindex.test** (perc=0.002105) Modell: **mboost** (family=Weibull, mstop=28, nu=0.474)
RS	5	0.447	Filter: **cindex.test** (perc=0.1761) Modell: **mboost** (family=Weibull, mstop=208, nu=0.1673)

Tabelle A.3: Konfigurationen für Datensatz „GSE4573“ nach B = 3000 MBO-Iterationen. Konkordanzindex ausgewertet auf unabhängigen Testdaten der äußeren Kreuzvalidierung.

Variante	Fold	C-Index	Hyperparameter
MBOK	1	0.552	Filter: **fmrmr** (perc=0.4814) Modell: **CoxBoost** (stepno=391, penalty=450.7, stepsize.factor=0.9818, sf.scheme=sigmoid)
MBOA	1	0.546	Filter: **fmrmr** (perc=0.4559) Modell: **CoxBoost** (stepno=137, penalty=33.88, stepsize.factor=0.1822, sf.scheme=sigmoid)
RS	1	0.555	Filter: **kratz.clinical** Modell: **mboost** (family=CoxPH, mstop=136, nu=0.2388)
MBOK	2	0.727	Filter: **cindex.train** (perc=0.6096) Modell: **CoxBoost** (stepno=185, penalty=529.9, stepsize.factor=1.03, sf.scheme=sigmoid)
MBOA	2	0.386	Filter: **kratz.clinical** Modell: **rfsrc** (ntree=8, mtry.ratio=0.8641, nodesize=6, splitrule=logrankscore)
RS	2	0.675	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=90, penalty=573.7, stepsize.factor=1.823, sf.scheme=linear)
MBOK	3	0.564	Filter: **kratz.clinical** Modell: **mboost** (family=CoxPH, mstop=42, nu=0.4242)
MBOA	3	0.619	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=69, penalty=774.8, stepsize.factor=1.24, sf.scheme=sigmoid)
RS	3	0.592	Filter: **kratz.clinical** Modell: **mboost** (family=CoxPH, mstop=303, nu=0.8563)

MBOK	4	0.596	Filter: **cindex.train** (perc=0.03369) Modell: **CoxBoost** (stepno=395, penalty=514.4, stepsize.factor=0.05191, sf.scheme=sigmoid)
MBOA	4	0.525	Filter: **fmrmr** (perc=0.9077) Modell: **CoxBoost** (stepno=22, penalty=729.8, stepsize.factor=1.804, sf.scheme=sigmoid)
RS	4	0.579	Filter: **fmrmr** (perc=0.6663) Modell: **mboost** (family=CoxPH, mstop=211, nu=0.6608)
MBOK	5	0.635	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=33, penalty=831, stepsize.factor=1.988, sf.scheme=sigmoid)
MBOA	5	0.627	Filter: **kratz** Modell: **CoxBoost** (stepno=19, penalty=896.3, stepsize.factor=1.77, sf.scheme=linear)
RS	5	0.565	Filter: **var** (perc=0.7135) Modell: **CoxBoost** (stepno=386, penalty=140.5, stepsize.factor=0.2536, sf.scheme=linear)

Tabelle A.4: Konfigurationen für Datensatz „GSE50081“ nach B = 3000 MBO-Iterationen. Konkordanzindex ausgewertet auf unabhängigen Testdaten der äußeren Kreuzvalidierung.

Variante	Fold	C-Index	Hyperparameter
MBOK	1	0.564	Filter: **var** (perc=0.2968) Modell: **CoxBoost** (stepno=215, penalty=201.3, stepsize.factor=0.7371, sf.scheme=sigmoid)
MBOA	1	0.570	Filter: **var** (perc=0.3094) Modell: **CoxBoost** (stepno=183, penalty=366.8, stepsize.factor=0.2888, sf.scheme=sigmoid)
RS	1	0.564	Filter: **coxscore** (perc=0.4951) Modell: **CoxBoost** (stepno=133, penalty=608.4, stepsize.factor=0.7011, sf.scheme=linear)

MBOK	2	0.379	Filter: **kratz.clinical** Modell: **mboost** (family=Weibull, mstop=8, nu=0.7932)
MBOA	2	0.632	Filter: **var** (perc=0.387) Modell: **CoxBoost** (stepno=287, penalty=93.79, stepsize.factor=0.1988, sf.scheme=sigmoid)
RS	2	0.557	Filter: **kratz.clinical** Modell: **rfsrc** (ntree=454, mtry.ratio=0.6987, nodesize=5, splitrule=logrankscore)
MBOK	3	0.592	Filter: **var** (perc=0.2215) Modell: **CoxBoost** (stepno=290, penalty=145.2, stepsize.factor=0.2417, sf.scheme=sigmoid)
MBOA	3	0.572	Filter: **var** (perc=0.04635) Modell: **CoxBoost** (stepno=389, penalty=399.5, stepsize.factor=0.1034, sf.scheme=sigmoid)
RS	3	0.531	Filter: **var** (perc=0.5629) Modell: **CoxBoost** (stepno=310, penalty=718.5, stepsize.factor=1.802, sf.scheme=linear)
MBOK	4	0.569	Filter: **var** (perc=0.3779) Modell: **CoxBoost** (stepno=315, penalty=58.45, stepsize.factor=0.1926, sf.scheme=sigmoid)
MBOA	4	0.513	Filter: **fmrmr** (perc=0.1367) Modell: **mboost** (family=CoxPH, mstop=489, nu=0.8972)
RS	4	0.533	Filter: **var** (perc=0.1273) Modell: **CoxBoost** (stepno=421, penalty=734.5, stepsize.factor=0.4352, sf.scheme=sigmoid)
MBOK	5	0.476	Filter: **clinical** Modell: **rfsrc** (ntree=13, mtry.ratio=0.4649, nodesize=10, splitrule=logrankscore)
MBOA	5	0.542	Filter: **var** (perc=0.02786) Modell: **CoxBoost** (stepno=249, penalty=678.7, stepsize.factor=0.4473, sf.scheme=sigmoid)

RS	5	0.457	Filter: **clinical** Modell: **rfsrc** (ntree=25, mtry.ratio=0.4226, nodesize=8, splitrule=logrankscore)

Tabelle A.5: Konfigurationen für Datensatz „GSE37745“ nach B = 3000 MBO-Iterationen. Konkordanzindex ausgewertet auf unabhängigen Testdaten der äußeren Kreuzvalidierung.

Variante	Fold	C-Index	Hyperparameter
MBOK	1	0.822	Filter: **cindex.test** (perc=0.09782) Modell: **CoxBoost** (stepno=419, penalty=827.2, stepsize.factor=0.08911, sf.scheme=sigmoid)
MBOA	1	0.804	Filter: **cindex.test** (perc=0.09503) Modell: **CoxBoost** (stepno=322, penalty=886.3, stepsize.factor=0.3806, sf.scheme=sigmoid)
RS	1	0.836	Filter: **cindex.train** (perc=0.6372) Modell: **CoxBoost** (stepno=152, penalty=687.1, stepsize.factor=0.2693, sf.scheme=sigmoid)
MBOK	2	0.609	Filter: **fmrmr** (perc=0.2194) Modell: **CoxBoost** (stepno=458, penalty=36.78, stepsize.factor=0.05796, sf.scheme=sigmoid)
MBOA	2	0.641	Filter: **fmrmr** (perc=0.1462) Modell: **CoxBoost** (stepno=148, penalty=929.9, stepsize.factor=0.2267, sf.scheme=sigmoid)
RS	2	0.573	Filter: **fmrmr** (perc=0.1592) Modell: **CoxBoost** (stepno=478, penalty=446, stepsize.factor=0.7966, sf.scheme=linear)
MBOK	3	0.784	Filter: **fmrmr** (perc=0.7068) Modell: **CoxBoost** (stepno=350, penalty=895.8, stepsize.factor=1.005, sf.scheme=sigmoid)
MBOA	3	0.719	Filter: **fmrmr** (perc=0.7646) Modell: **CoxBoost** (stepno=4, penalty=196.4, stepsize.factor=0.8241, sf.scheme=sigmoid)

RS	3	0.688	Filter: **coxscore** (perc=0.5829) Modell: **mboost** (family=CoxPH, mstop=117, nu=0.6637)
MBOK	4	0.689	Filter: **cindex.train** (perc=0.8761) Modell: **CoxBoost** (stepno=123, penalty=380.6, stepsize.factor=0.652, sf.scheme=sigmoid)
MBOA	4	0.738	Filter: **cindex.train** (perc=0.03988) Modell: **CoxBoost** (stepno=226, penalty=846.5, stepsize.factor=0.5784, sf.scheme=sigmoid)
RS	4	0.685	Filter: **cindex.train** (perc=0.6625) Modell: **CoxBoost** (stepno=338, penalty=227.1, stepsize.factor=0.01932, sf.scheme=linear)
MBOK	5	0.586	Filter: **kratz.clinical** Modell: **rfsrc** (ntree=325, mtry.ratio=0.4862, nodesize=1, splitrule=logrankscore)
MBOA	5	0.665	Filter: **coxscore** (perc=0.02491) Modell: **CoxBoost** (stepno=32, penalty=528.7, stepsize.factor=0.4176, sf.scheme=sigmoid)
RS	5	0.639	Filter: **cindex.test** (perc=0.9816) Modell: **cvglmnet** (alpha=0.2944)

Tabelle A.6: Konfigurationen für Datensatz „GSE31210“ nach B = 3000 MBO-Iterationen. Konkordanzindex ausgewertet auf unabhängigen Testdaten der äußeren Kreuzvalidierung.

Variante	Fold	C-Index	Hyperparameter
MBOK	1	0.620	Filter: **var** (perc=0.1301) Modell: **CoxBoost** (stepno=240, penalty=765.5, stepsize.factor=0.6565, sf.scheme=sigmoid)
MBOA	1	0.622	Filter: **var** (perc=0.1246) Modell: **CoxBoost** (stepno=405, penalty=130.5, stepsize.factor=0.8564, sf.scheme=sigmoid)

RS	1	0.614	Filter: **var** (perc=0.301) Modell: **CoxBoost** (stepno=484, penalty=606.5, stepsize.factor=0.7012, sf.scheme=linear)
MBOK	2	0.620	Filter: **fmrmr** (perc=0.5158) Modell: **CoxBoost** (stepno=228, penalty=465.9, stepsize.factor=0.1909, sf.scheme=sigmoid)
MBOA	2	0.665	Filter: **kratz.clinical** Modell: **mboost** (family=CoxPH, mstop=174, nu=0.6382)
RS	2	0.675	Filter: **cindex.train** (perc=0.3256) Modell: **CoxBoost** (stepno=176, penalty=539.2, stepsize.factor=0.5668, sf.scheme=sigmoid)
MBOK	3	0.624	Filter: **kratz.clinical** Modell: **mboost** (family=CoxPH, mstop=171, nu=0.8775)
MBOA	3	0.630	Filter: **kratz.clinical** Modell: **mboost** (family=CoxPH, mstop=278, nu=0.5276)
RS	3	0.628	Filter: **kratz.clinical** Modell: **mboost** (family=CoxPH, mstop=495, nu=0.2717)
MBOK	4	0.667	Filter: **var** (perc=0.07871) Modell: **CoxBoost** (stepno=444, penalty=405.2, stepsize.factor=0.1782, sf.scheme=sigmoid)
MBOA	4	0.587	Filter: **coxscore** (perc=0.2639) Modell: **CoxBoost** (stepno=380, penalty=158.3, stepsize.factor=0.4055, sf.scheme=sigmoid)
RS	4	0.659	Filter: **var** (perc=0.1656) Modell: **CoxBoost** (stepno=313, penalty=745.7, stepsize.factor=0.4385, sf.scheme=sigmoid)
MBOK	5	0.737	Filter: **var** (perc=0.09269) Modell: **CoxBoost** (stepno=161, penalty=999.3, stepsize.factor=0.177, sf.scheme=sigmoid)

MBOA	5	0.730	Filter: **var** (perc=0.05089) Modell: **CoxBoost** (stepno=436, penalty=957.8, stepsize.factor=0.01072, sf.scheme=sigmoid)
RS	5	0.705	Filter: **var** (perc=0.646) Modell: **CoxBoost** (stepno=168, penalty=737.9, stepsize.factor=0.08956, sf.scheme=sigmoid)

Tabelle A.7: Konfigurationen für Datensatz „GSE30219“ nach B = 3000 MBO-Iterationen. Konkordanzindex ausgewertet auf unabhängigen Testdaten der äußeren Kreuzvalidierung.

Variante	Fold	C-Index	Hyperparameter
MBOK	1	0.649	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=10, penalty=167.5, stepsize.factor=0.3044, sf.scheme=sigmoid)
MBOA	1	0.569	Filter: **var** (perc=0.1257) Modell: **CoxBoost** (stepno=371, penalty=8.333, stepsize.factor=0.9317, sf.scheme=sigmoid)
RS	1	0.653	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=170, penalty=785.5, stepsize.factor=0.5228, sf.scheme=sigmoid)
MBOK	2	0.611	Filter: **kratz.clinical** Modell: **rfsrc** (ntree=524, mtry.ratio=0.007018, nodesize=10, splitrule=logrankscore)
MBOA	2	0.606	Filter: **kratz.clinical** Modell: **rfsrc** (ntree=986, mtry.ratio=0.1629, nodesize=1, splitrule=logrankscore)
RS	2	0.587	Filter: **coxscore** (perc=0.9698) Modell: **mboost** (family=CoxPH, mstop=229, nu=0.9217)
MBOK	3	0.633	Filter: **var** (perc=0.3887) Modell: **CoxBoost** (stepno=18, penalty=600, stepsize.factor=0.791, sf.scheme=sigmoid)

MBOA	3	0.635	Filter: **var** (perc=0.3515) Modell: **CoxBoost** (stepno=48, penalty=514.9, stepsize.factor=1.129, sf.scheme=sigmoid)
RS	3	0.586	Filter: **var** (perc=0.1456) Modell: **CoxBoost** (stepno=487, penalty=784.4, stepsize.factor=0.7804, sf.scheme=linear)
MBOK	4	0.659	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=14, penalty=67.84, stepsize.factor=0.6561, sf.scheme=sigmoid)
MBOA	4	0.635	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=434, penalty=777.4, stepsize.factor=0.4536, sf.scheme=sigmoid)
RS	4	0.645	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=73, penalty=636.7, stepsize.factor=0.5257, sf.scheme=sigmoid)
MBOK	5	0.613	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=52, penalty=147.1, stepsize.factor=1.783, sf.scheme=sigmoid)
MBOA	5	0.622	Filter: **coxscore** (perc=0.4979) Modell: **CoxBoost** (stepno=18, penalty=310.4, stepsize.factor=1.872, sf.scheme=sigmoid)
RS	5	0.605	Filter: **kratz.clinical** Modell: **CoxBoost** (stepno=383, penalty=270.6, stepsize.factor=1.211, sf.scheme=linear)

Tabelle A.8: Konfigurationen für Datensatz „Shedden“ nach B = 3000 MBO-Iterationen. Konkordanzindex ausgewertet auf unabhängigen Testdaten der äußeren Kreuzvalidierung.

Datensatz	Männlich	Weiblich
GSE14814	67	23
GSE4573	82	47
GSE50081	98	83
GSE37745	105	89
GSE31210	105	121
GSE30219	228	40
Shedden	223	219
Insgesamt	908	622

Tabelle A.9: Übersicht über die Kovariable Geschlecht: Absolute Häufigkeiten.

Datensatz	squamous	adeno	largecell	other
GSE14814	52	28	10	0
GSE4573	129	0	0	0
GSE50081	43	127	7	4
GSE37745	64	106	24	0
GSE31210	0	226	0	0
GSE30219	60	85	55	68
Shedden	0	442	0	0
Insgesamt	348	1014	96	72

Tabelle A.10: Übersicht über die Kovariable Histologie: Absolute Häufigkeiten der Tumortypen „squamos“ (Plattenepithelkarzinom), „adeno“ (Adenokarzinom), „largecell“ (Großzelliges Karzinom) und „other“ (andere oder unbestimmt).

Datensatz	$\min(x)$	$x_{0.25}$	$x_{0.5}$	$\bar{x}$	$x_{0.75}$	$\max(x)$
GSE14814	38.2	57.4	63.25	62.07	67.35	81.3
GSE4573	42	60	68	67.41	75	91
GSE50081	40.16	63	69.77	68.78	74.16	87.93
GSE37745	39	57	65	63.66	70	84
GSE31210	30	55	61	59.58	65	76
GSE30219	15	54	62	61.03	70	84
Shedden	33	58	65	64.39	72	87
Insgesamt	15	57	64	63.64	70.73	91

Tabelle A.11: Übersicht über die Kovariable Alter: Minimum, unteres Quantil, Median, arithmetisches Mittel, oberes Quantil und Maximum.

B Grafiken

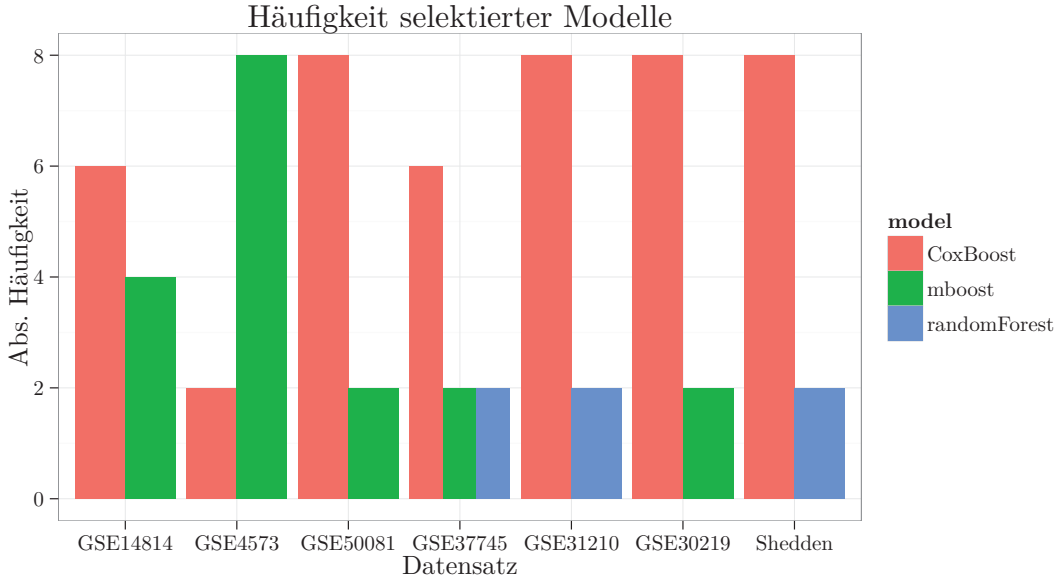

Abbildung B.1: Übersicht über die Häufigkeit final selektierter Modelle der klassischen MBO-Variante MBOK nach 3000 Iterationen.

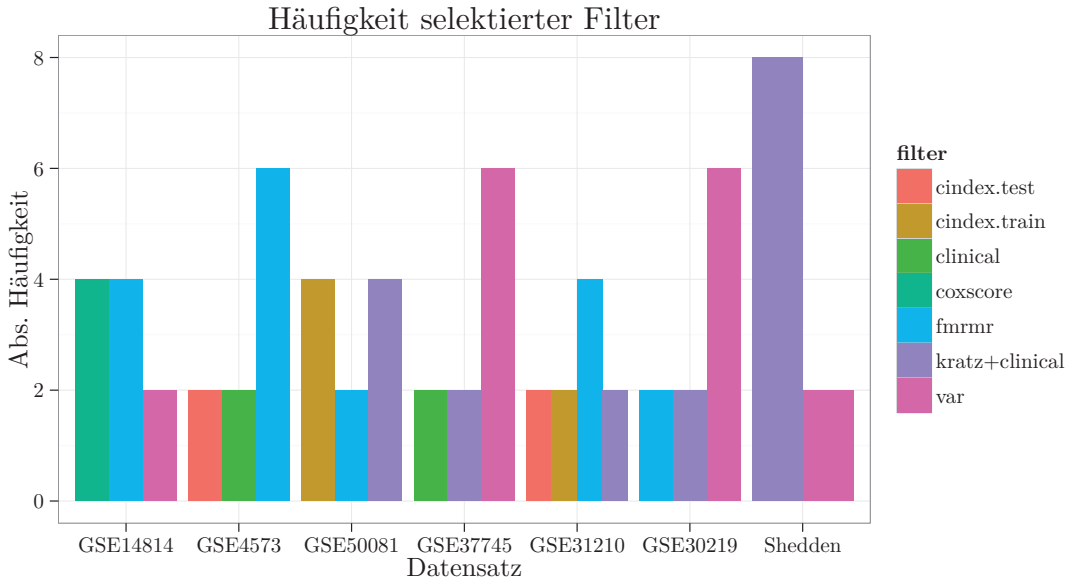

Abbildung B.2: Übersicht über die Häufigkeit final selektierter Filter der klassischen MBO-Variante MBOK nach 3000 Iterationen.

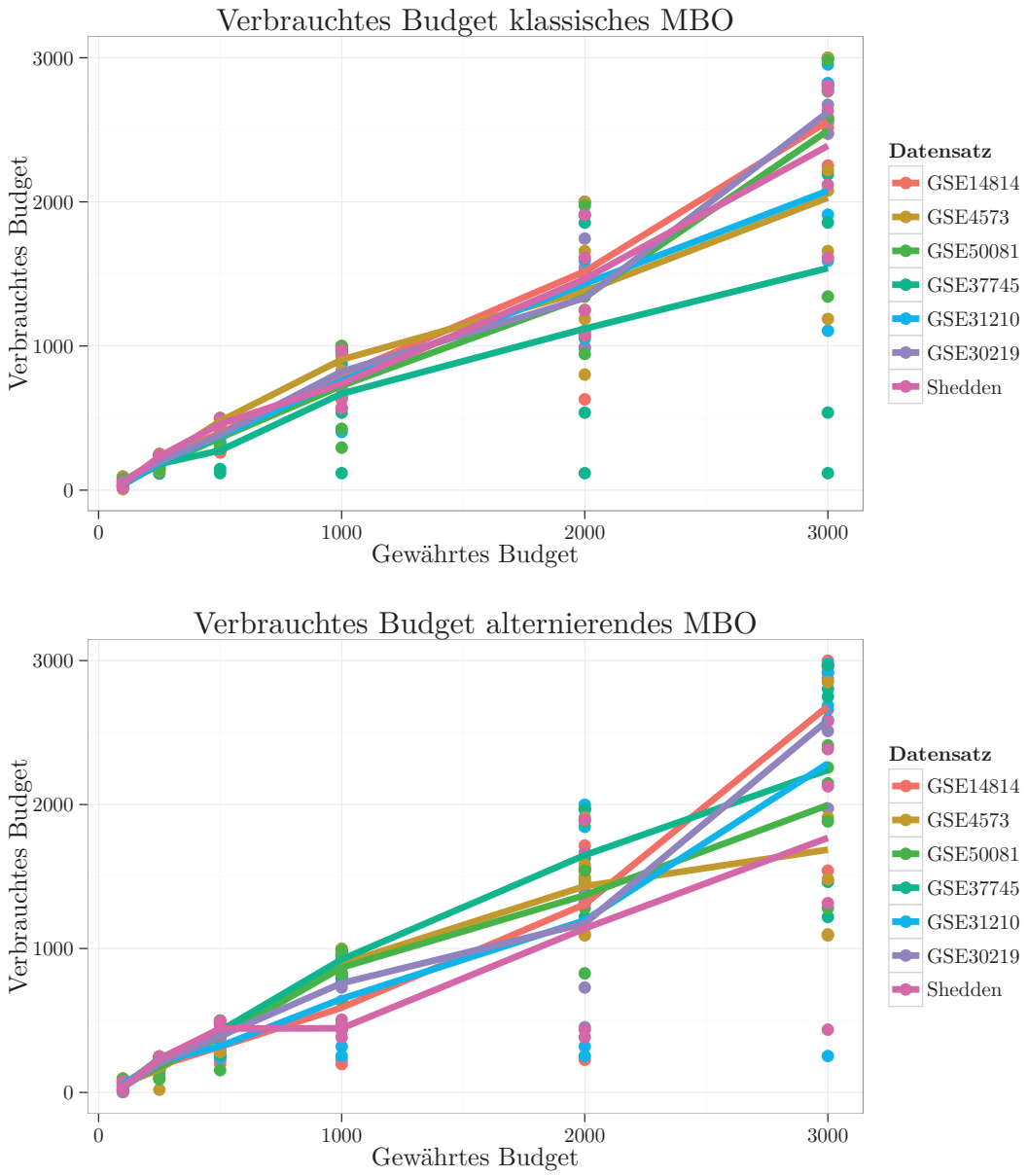

Abbildung B.3: Gewährtes Budget gegen tatsächlich benötigtes Budget der MBO Varianten MBOK (oben) und MBOA (unten). Optimierungspfad wurde nach 100 (initiales LHS-Design), 250, 500, 1000, 2000 und 3000 Iterationen analysiert (x-Achse), jeweils die beste Konfiguration bis zu dieser Iteration bestimmt und deren Iterationszahl auf der y-Achse abgetragen.

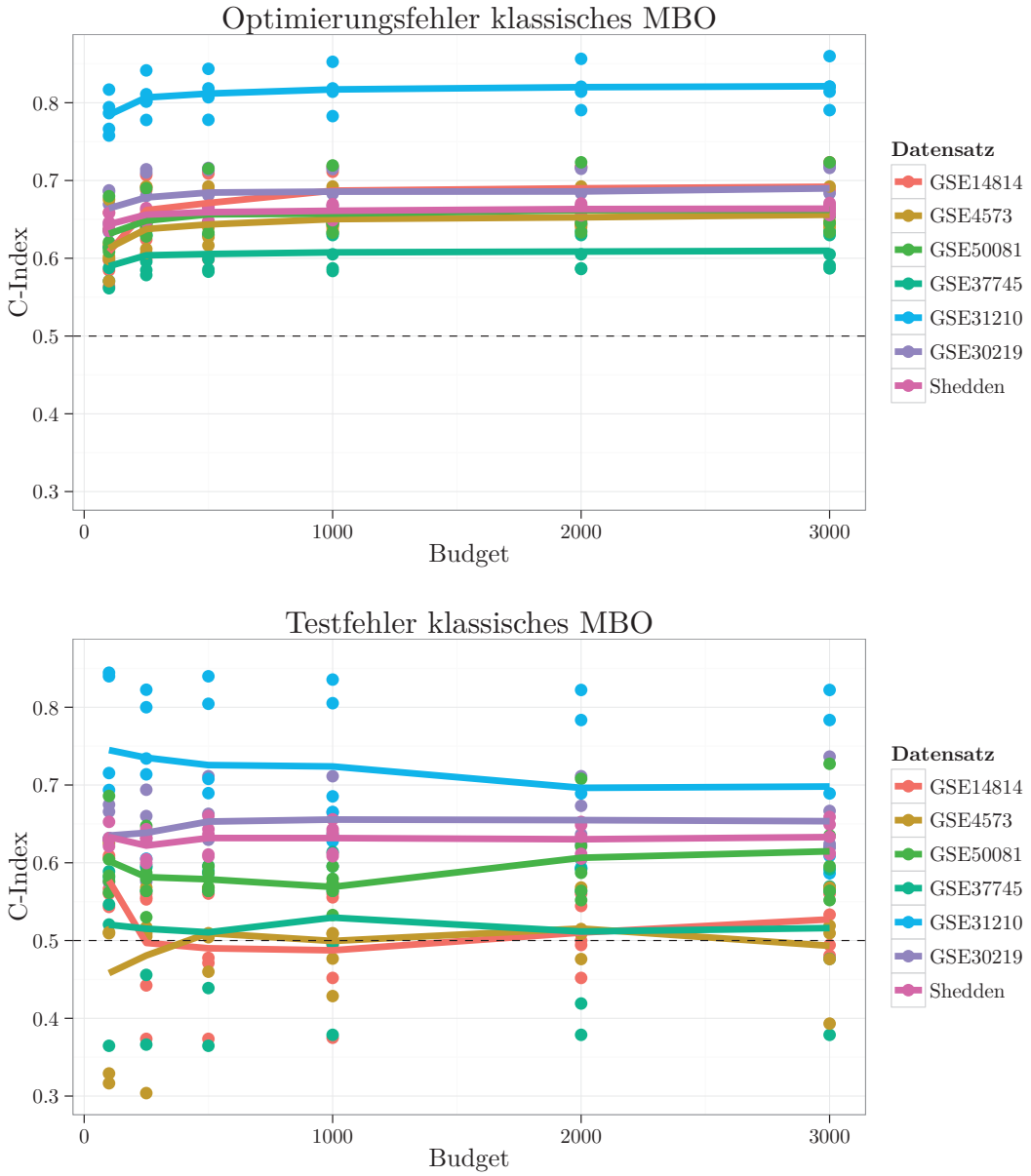

Abbildung B.4: Optimierungs- und Testfehler nach 100 (initiales LHS-Design), 250, 500, 1000, 2000 und 3000 Iterationen des klassischen MBOs.

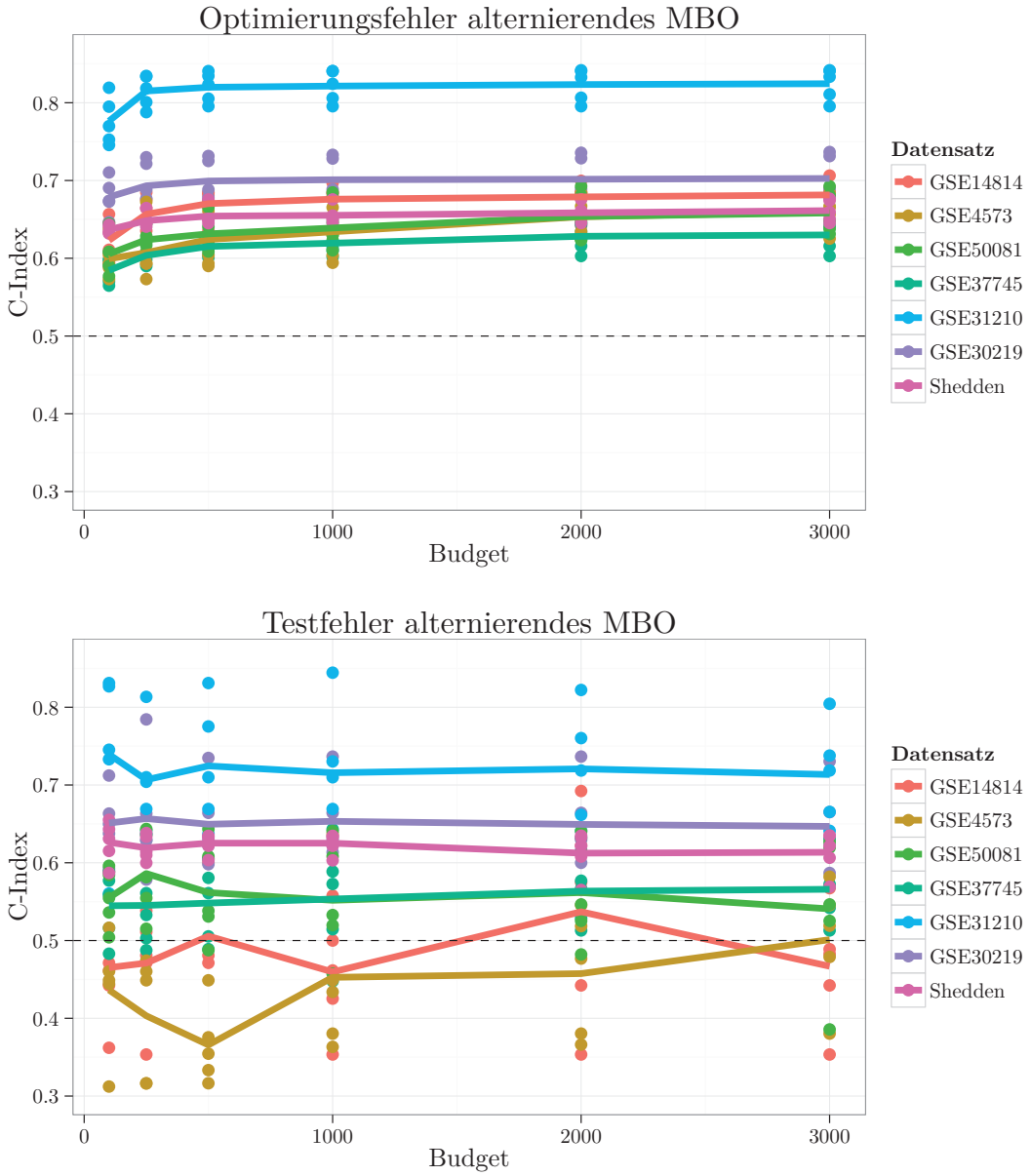

Abbildung B.5: Optimierungs- und Testfehler nach 100 (initiales LHS-Design), 250, 500, 1000, 2000 und 3000 Iterationen des alternierenden MBOs.

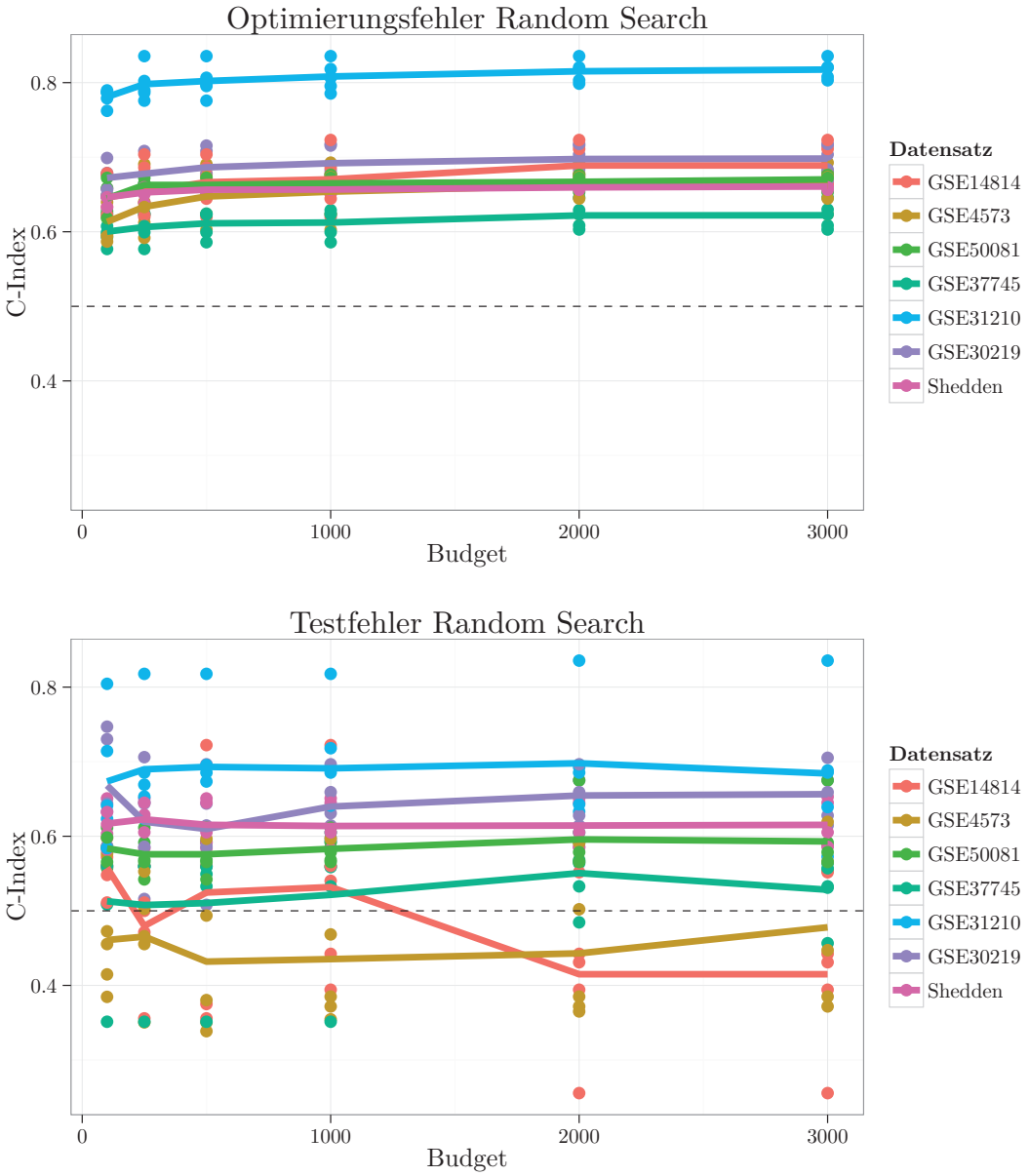

Abbildung B.6: Optimierungs- und Testfehler nach 100 (nur initiales LHS-Design), 250, 500, 1000, 2000 und 3000 Iterationen der Random Search.